Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anton Semprimožnik

Analiza zmogljivosti mobilnih spletnih brskalnikov pri izrisovanju videa

MAGISTRSKO DELO
ŠTUDIJSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

izr. prof. dr. Iztok Lebar Bajec
MENTOR

Ljubljana, 2016

© 2016, Anton Semprimožnik

Rezultati magistrskega dela so intelektualna lastnina avtorja ter Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov magistrskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

IZJAVA O AVTORSTVU MAGISTRSKEGA DELA

Spodaj podpisani izjavljam, da sem avtor dela, da slednje ne vsebuje materiala, ki bi ga kdorkoli predhodno že objavil ali oddal v obravnavo za pridobitev naziva na univerzi ali drugem visokošolskem zavodu, razen v primerih, kjer so navedeni viri.
S svojim podpisom zagotavljam, da:

- sem delo izdelal samostojno pod mentorstvom izr. prof. dr. Iztoka Lebarja Bajca,
- so elektronska oblika dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko in
- soglašam z javno objavo elektronske oblike dela v zbirki "Dela FRI".

— Anton Semprimožnik, Ljubljana, december 2016.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anton Semprimožnik

Analiza zmogljivosti mobilnih spletnih brskalnikov pri izrisovanju videa

POVZETEK

V magistrskem delu smo analizirali in primerjali zmogljivost zaporedne in vzporedne implementacije dekodirnika videa. Za implementacijo smo uporabili jezik Javascript. Pri zaporedni implementaciji smo za dekodiranje uporabili glavno nit, medtem ko smo pri vzporedni uporabili dodatno nit.

V delu smo podrobno preučili vpliv vzporednosti na razbremenitev glavne niti. Pri zaporedni implementaciji je bila glavna nit v celoti obremenjena z dekodiranjem. Pri vzporedni implementaciji smo dekodiranje izvajali na vzporedni niti, zaradi česar je bila glavna nit popolnoma razbremenjena ali obremenjena z operacijami upravljanja uporabniškega vmesnika, ki jih ne moremo izvajati vzporedno. Meritve smo izvajali v namiznih spletnih brskalnikih in v spletnih brskalnikih desetih najbolj uporabljanih mobilnih naprav. Pri tem smo poizkušali ugotoviti, kakšen vpliv ima vzporednost na hitrost dekodiranja in izrisovanje videa.

Ugotovili smo, da je uporaba vzporednosti smiselna le v primerih, kadar je glavna nit močno obremenjena z operacijami za upravljanje uporabniškega vmesnika. V primeru, da z odlaganjem bremena na vzporedno nit glavno popolnoma razbremenimo, na zmogljivosti aplikacije ne pridobimo.

Ključne besede: realno-časovna grafika, mobilni splet, zmogljivost mobilnih naprav, vzporednost

University of Ljubljana
Faculty of Computer and Information Science

Anton Semprimožnik

Performance analysis of video rendering in mobile web browsers

ABSTRACT

In this thesis we analyzed and compared the performance of a single threaded and a parallel video decoder both implemented in Javascript language. In the single threaded implementation we use the main thread to decode a video. In the parallel version we use an additional thread.

We analyzed the effect of parallelism on the main thread's performance. In the single threaded implementation the main thread was fully utilized with video decoding. In the parallel version we were decoding the video on an additional parallel thread with the main thread empty or partially utilized with user interface tasks that cannot be executed in parallel. We measured the performance in desktop web browsers and on the ten most common mobile devices on the market. We evaluated the effect of parallelism on the decoding and rendering speed.

We show that parallelism improves performance when the main thread executes heavy tasks related to the user interface. In the case when the main thread's load is fully offloaded to the parallel thread the performance does not improve.

Key words: real-time graphics, mobile web, mobile device performance, parallelism

ZAHVALA

Zahvaljujem se vsem, ki so mi na kakršenkoli način pomagali in me spodbudili k izdelavi magistrskega dela. Zahvalil bi se mentorju izr. prof. dr. Iztoku Lebarju Bajcu za podporo in odlično sodelovanje.

— Anton Semprimožnik, Ljubljana, december 2016.

KAZALO

1 Uvod

1.1 Motivacija

Izmed vseh uporabnikov spleta je 80 % takšnih, ki za dostop uporabljajo svoj mobilni telefon. Mobilne naprave današnjega časa imajo velike zaslone, zmogljive vmesnike ter so visoko optimizirane za brskanje po spletu. Po statistikah je število uporabnikov mobilnega spleta preseglo tiste s fiksnim dostopom do spleta (namizni računalniki) že leta 2014. Mobilni splet ni več področje, ki ga lahko spregledamo, in podjetniki, ki svojih strank ne morejo doseči preko mobilnih naprav ali pa jih, vendar je njihova storitev slaba, hitro izgubljajo proti tekmecem.

V vedno bolj s spletom povezanem svetu je Javascript vsepovsod. V spletnih brskalnikih, zalednih sistemih kot tudi namiznih ali angl. *native* mobilnih aplikacijah je sestavni del operacijskih sistemov in podobno. Javascript se ne uporablja le kot izvorni jezik aplikacij, ampak tudi kot ciljni jezik spletnih aplikacij, ki se razvijajo v drugih jezikih. Moderni brskalniki ponujajo programerjem celo množico novih funkcionalnosti, ki so bile do sedaj na voljo v tradicionalnih programskih jezikih. Razširitve, kot so različni video in zvočni predvajalniki, so danes implementirane v brskalnikih. Z novimi funkcionalnostmi

brskalnikov naraščajo možnosti izdelave aplikacij, ki tečejo tako v namiznih kot mobilnih brskalnikih na osnovi enake kode. Brskalniki postajajo zelo pomembna platforma, ki omogoča razvoj široko-platformnih (angl. *cross-platform*) aplikacij brez nepotrebnega razvoja različic aplikacij za različne ciljne sisteme.

Velik napredek v zmogljivosti programskih okolij Javascripta (angl. *Javascript engines*), kot tudi samega jezika, omogoča realizacijo računsko zahtevnih in interaktivno bogatih aplikacij. Velika težava Javascripta, napram ostalim jezikom, je pomanjkanje mehanizmov, ki bi omogočali njegovo vzporedno izvajanje. Dinamičnost jezika prav tako onemogoča razvoj visoko učinkovitih pretvornikov (angl. *compiler*), ki bi optimizirali izvajanje znotraj odjemalca, kaj šele učinkovito vzporednost.

V časih, ko ima vsaka mobilna naprava večjedrni procesor in ko spletne aplikacije in dostop do spleta predstavljajo večino naših interakcij s spletom na mobilnih napravah, ostajajo naše naprave popolonoma neizkoriščene in uporabniki s slabo izkušnjo. Tradicionalni Javascript ima eno-nitni model izvajanja, kar predstavlja oviro za izkoriščanje računske moči več-jedrnih procesorjev, bodisi namiznih računalnikov ali mobilnih naprav. Zaradi visoke potrebe po vzporednosti, a nezmožnosti slednje zaradi dinamičnosti jezika, je z namenom reševanja zahtevnejših problemov vzporednost implementirana z uporabo spletnih delavcev (angl. *Web Workers*[1]) na nivoju brskalnika. Ti omogočajo izvajanje skript v nitih, ločenih od glavne. Niti ne omogočajo upravljanja z uporabniških vmesnikom, risanje grafike na uporabniški vmesnik in podobno. Komuniciranje med glavno nitjo in nitmi spletnih delavcev je omejeno in počasno, saj je pomnilniški prostor v celoti izoliran/ločen.

1.2 Pregled sorodnih del

Vzporednost na spletu je nov in neraziskan pojem, študij in analiz na tem področju je malo. Opravljenih je nekaj teoretičnih raziskav pohitritev na nivoju jezika in alternativ z uporabo HTML5 spletnih delavcev. Optimizacije v modernih okoljih nas pripeljejo do teoretičnih hitrosti blizu rešitvam, ki se izvajajo v programskih jezikih, kot je C — a so okolja, v katerih se teste izvaja, večinoma eksperimentalna in zaradi mnogih omejitev neuporabna v praksi kot morebitno nadomestilo današnjih brskalnikov. Noben članek se ne ukvarja z vprašanjem, kakšen vpliv imajo vzporedne predelave na glavno nit in

[1] Web Workers: html5rocks.com/en/tutorials/workers/basics

delovanje uporabniškega vmesnika.

Članek o odlaganju računskih operacij [1] predlaga pohitritev računsko zahtevnih operacij z odlaganjem procesiranja v oblak. Kot prednosti načina izpostavljajo varčevanje z energijo in odložitev bremena na močnejše naprave v zaledju. Kot testno aplikacijo navajajo 3D igro, ki prikazuje interakcije med kockami na zaslonu. Nit brskalnika preračunava vse računske operacije, detekcijo trkov in fiziko igre, gravitacijo in podobno. Uporabili so eno nit, ki periodično posodobi vse objekte. V članku trdijo, da je prenos logike iz niti v oblak izboljšala število izrisanih sličic na sekundo za faktor 1,4 do 2,3, kar pomeni 7 sličic na sekundo pri lokalnem preračunavanju in 15 z odlaganjem v oblak. Poizkus je testiran na zmogljivih osebnih računalnikih. V članku se zavedajo in poudarjajo, da lahko slabo omrežje močno vpliva na rezultate, kar poudarja pomembnost uporabe vzporednosti v brskalniku. Način se izkaže za uporabnega pri dolgotrajnih in računsko zelo zahtevnih operacijah, v primeru grafike v realnem času na mobilnih napravah pa je pristop zaradi izdatne omrežne komunikacije neprimeren, saj je izvajanje močno odvisno od nihajočih hitrosti mobilnih omrežij in večje latence.

Članek o vzporedni implementaciji interaktivnih animacij [2] opisuje uporabo vzporednosti v arkadni igri, ki za vsak korak animacije potrebuje 0,3 sekunde (3,3 sličice na sekundo). Program posodablja večje število objektov (iskanje trkov, posodabljanje pozicije ...), medtem ko glavna nit skrbi za celoten izris in kreiranje vsebine. Avtorji opozarjajo na nezmožnost upravljanja elementov DOM v uporabljeni tehnologiji. Evalvacija oceni pohitritev pri različnem številu objektov, ki jih niti posodabljajo. V okviru članka ugotovijo, da vzporednost pospeši izvajanje za 30 %, počasno risanje interaktivne grafike pa reši le do neke mere. Kot morebitno izboljšavo predlagajo večjo razbremenitev glavne niti z uporabo spletnih delavcev, saj glavna nit riše na zaslon in je odgovorna za odzivnost spletnega vmesnika. Hitrost risanja na zaslon je za grafiko v realnem času popolnoma nezadovoljiva.

Članek [3] se, kot predhodni, ukvarja z enako idejo odlaganja dela v oblak. Iz meritev ugotovi, da omrežne zakasnitve pri uporabi vzporednosti, ki skrbi za odlaganje dela v oblak, doprinesejo slabih 15 % napram uporabi glavne niti. Uspešnost testira na zmogljivih osebnih računalnikih in mobilnih napravah, kjer ugotovi, da te zaradi manjše zmogljivosti doživijo do 3-kratno pohitritev pri odlaganju bremena v oblak.

Verdu in Pajuelo z Barcelonske univerze [4] se ukvarjata s skalabilnostjo Javascript aplikacij pri uporabi vzporednosti. Rezultati pokažejo, da je optimalna izvedba močno

odvisna od arhitekture CPE, verzije brskalnika in podobno. Ugotavljata, da je uporaba manjšega števila niti v večni primerov hitrejša kot uporaba večjega, sočasno poganjanje drugih aplikacij pa naj bi imelo velik vpliv na zmogljivost. Testiranje je izvedeno na zmogljivih računalnikih.

Martinsen, Grahn in Isberg [5] implementirajo TLS (angl. *Thread Level Speculation*) v okolju Squirrelfish[2], ki dovoli vzporedno izvajanje funkcij. Squirrelfish je testni prepis jedra Javascript v bitni interpreter. Pristop TLS dovoli izločanje posameznih delov sekvenčnega programa (posameznih operacij) in njihovo vzporedno izvedbo glede na izvedeno špekulacijo. Potek zaporednih programov je vedno natančno določen, tako tudi dostop do podatkov, kjer pri preoblikovanju v vzporedno izvajanje posamezne akcije kršijo zaporedni dostop do le-teh. Avtorji primerjajo izvajanje implementacije TLS, okolja Google V8 in okolja Squirrelfish z običajno sekvenčno izvedbo. Pri vključenem TLS-ju dosežejo pohitritve za faktor 8. To nakazuje na velik potencial vzporednosti v jeziku. Število zahtevanih vrnitev na prejšnje stanje programa (angl. *rollback*) je še vedno majhno v primerjavi s številom uspešnih vzporednih izvedb. Ugotavljajo, da se lahko istočasno izvaja veliko število niti. Sedem od 50 spletnih aplikacij je izvajalo tudi po 50 niti hkrati. Za realno uporabo je ta princip nezadovoljiv, saj je Squirrelfish le testno okolje z mnogimi omejitvami in posledično močno omejena verzija današnjih brskalnikov. Članek dobro razišče in pokaže teoretične možnosti vzporednosti v jeziku, a se ta v praksi v tej smeri ne razvija.

Članek [6] iz leta 2010 se s pohitritvijo aplikacij Javascript ukvarja na drugačen način (vzporednost na spletu ob nastanku članka še ni bila podprta s strani brskalnikov). Teoretično analizira posnete sledi izvajanja računsko zahtevnih spletnih aplikacij (npr. simulator tekočin), kjer uporabijo nizkonivojski, od podatkov odvisen pristop z namenom iskanja potencialne vzporednosti. V članku ugotovijo, da je vzporednost izvedljiva in smiselna. Večina zank, napisanih v programskem jeziku Javscript, sicer ne iterira dovolj pogosto, iteracije pa si niso dovolj podobne, da bi lahko bile učinkovito izvedene v vzporednem načinu. Priložnosti za vzporednost se pojavijo pri računsko zahtevnih funkcijah, kjer opravičijo morebitno dodatno manipulacijo za potrebno izvajanja vzporednosti. Sklepajo, da ima jezik velik potencial za vzporednost, ki bi zmanjšala čas izvajanja in povečala odzivnost. Vzporednost bi nam omogočila boljšo uporabo več-jedrnih, nizkoenergijskih procesorjev na mobilnih napravah, hitrejše aplikacije pa bi programerjem

[2]SquirrelFish: trac.webkit.org/wiki/SquirrelFish

omogočile izvajanje računsko zahtevnih operacij ter bolj bogate in odzivne aplikacije. Raziskujejo odvisne tipe in obnašanje iteracij za boljše razumevanje morebitnih pohitritev. Rezultati kažejo na teoretične pohitritve, v povprečju za faktor 8,9. Vzporedna implementacija funkcij se pokaže za bolj učinkovito napram vzporednosti zank. V članku tudi ugotavljajo, da večina referenc kaže na virtualni register in ne v razpršene tabele (angl. *hash table*), ki bi pohitrile dostop do podatkov. Testiranje so izvedli na zmogljivih namiznih računalnikih.

Po širokem pregledu področja, izkušnjah iz realnih problemov ter pregledu mnogih (teoretičnih) rešitev in raziskav smo ugotovili, da zanesljivih analiz zmogljivosti in učinkovitosti vzporednosti v mobilnih brskalnikih ni. Prav tako ne najdemo jasnega odgovora na vprašanje, ali je v aplikaciji vzporednost, ki ni rešena na nivoju jezika, sploh smiselno uporabiti.

Ugotavljamo, da:

- ni analiz pohitritev pri uporabi vzporednosti v realnih okoljih,
- ni informacij ali meritev o vplivih vzporednosti na glavno nit, ki ostaja edina zmožna upravljati grafiko in uporabniško interakcijo,
- so obstoječe primerjave nepopolne ali opravljene le na zmogljivih namiznih računalnikih,
- rešitev, ki jih predlagajo analize ni mogoče realizirati,
- analize ne upoštevajo značilnosti Javascripta in posebnosti delovanja brskalnikov,
- ni primerjav in analiz, ki bi kazale na morebitne razlike med različnimi napravami,
- ni jasnih primerjav med delovanjem različnih brskalnikov,
- ni primerjav vpliva relacije med brskalnikom z napravo in OS,
- obstaja vrsta teoretičnih raziskav, ki se osredotoča na vzporednost jezika Javascript, kjer pa zaradi dinamičnosti jezika podpore vzporednosti ni na vidiku.

1.3 Cilji in pričakovani rezultati

V nalogi analiziramo in primerjamo zmogljivost eno-nitne z vzporedno implementacijo video dekodirnika, razvitega v jeziku Javascript. Osredotočamo se na računsko zmogljivost pri uporabi vzporednosti ter vpliv na glavno nit brskalnika in njegovo nemoteno

delovanje pri veliki obremenitvi. Pričakujemo razlike v zmogljivosti med brskalniki v odvisnosti od operacijskih sistemov in celo znamk naprav, na katerih brskalniki tečejo. Z uporabo vzporednosti skušamo izboljšati zmogljivost mobilnih aplikacij in dokazati, da je s pravilnimi pristopi lahko doseči največji izkoristek kapacitet, ki so nam na voljo, brez vpliva na uporabniško izkušnjo. Trdimo, da je za slabo zmogljivost aplikacij kriv predvsem napačen pristop in nepoznavanje posebnosti programskega jezika ter delovanja brskalnikov.

S pomočjo razvitega Javascript video dekodirnika bomo izmerili in analizirali zmogljivost brskalnikov mobilnih naprav pri uporabi glavne niti ali vzporednosti. Vzporedna implementacija video dekodirnika bo predstavljena kot alternativa najbolj pogostemu, a slabemu pristopu reševanja grafike v realnem času. Na brskalnikih mobilnih naprav se celotno procesiranje, povezano s pripravo vsebine, ki jo izrisujemo na zaslon, izvaja na glavni niti. Časovno potratne operacije se alternativno z uporabo vzporednosti odlagajo v oblak. Ta način je zaradi potrebe po omrežni komunikaciji primeren le za dalj časa trajajoče računsko zahtevne operacije, nalaganje glavnine dela glavni niti pa neučinkovito, saj slednja skrbi za vse procese, ki tečejo v brskalniku.

Glavne težave, s katerimi se na tem področju srečujemo, so mnoge omejitve pri uporabi vzporednosti, slabša strojna oprema mobilnih naprav in omejen dostop do slednje. Dekodiranje videa ali druge računsko zahtevne operacije so zaradi mnogih omejitev tehnologije, ki omogoča vzporednost, običajno implementirane le na glavni niti. Ta je zadolžena za vse operacije, ki tečejo v brskalniku, in je tako za izvajanje težkih računskih operacij premalo zmogljiva. Dekodiranje videa na glavni niti je računsko zelo potratno, neučinkovito in močno vpliva na odzivnost brskalnika. Zanimivo bo izmeriti pohitritve pri uporabi vzporednosti, povečanje števila izrisanih sličic na sekundo in izboljšanje odzivnosti vmesnika, saj bomo močno razbremenili glavno nit.

1.4 Pomembne novosti

Do danes (oktober 2016) je bila uporaba dekodirnikov, razvitih v jeziku Javascript, edina možnost za samodejno predvajanje videa v mobilnih brskalnikih sodobnih operacijskih sistemov. Operacijski sistem Android je edini omogočal predvajanje videa z video elementom, implementiranim na nivoju brskalnika, a je le-ta zahteval uporabniško interakcijo. Zaradi zahtevne implementacije dekodirnikov in zmogljivosti jezika Javascript so

bili takšni dekodirniki v praksi skorajda neobstoječi. Aktivno so se rešitve problema te vrste pričele uporabljati v prvi polovici leta 2015, največ zaradi pritiska velikih ponudnikov spletnih vsebin ter oglaševalcev in hitro naraščajoče vloge in pomembnosti videa na mobilnih napravah. Z novimi različicami brskalnikov obe najbolj množični platformi omogočata sistemsko (samodejno) predvajanje videa brez uporabniške interakcije. Sprememba izloči dve veliki omejitvi, ki smo si jih v magistrskem delu z razvitim dekodirnikom v Javascript jeziku zadali reševati — samodejno predvajanje videa ter nezmožnost predvajanja videa znotraj vsebine strani (angl. *plays inline*[3]). Video dekodirnik v magistrskem delu tako služi zgolj kot breme izvedenih analiz.

1.5 Strokovni prispevki

Strokovni prispevek magistrskega dela bodo podrobne meritve in analiza zmogljivosti brskalnikov vsaj desetih različnih, najbolj pogostih mobilnih naprav na trgu pri uporabi glavne niti ali vzporednosti za dekodiranje videa. Poiskali bomo odgovore na vprašanja, ki smo si jih zastavili v poglavju 1.1.

Strokovni prispevek bo v okviru testne aplikacije, uporabljene za meritve, razvit običajni in večnitni Javascript video dekodirnik, kjer bo potrebno poiskati alternativo nezmožnosti uporabe elementov DOM v vzporednem sistemu. Izmerili in analizirali bomo zmogljivost na podlagi razvitega video dekodirnika, ki bo v celoti samozadosten in bo slonel le na brskalniku mobilne naprave brez uporabe zunanjih virov za pripravo vsebine.

1.6 Metodologije

Evalvacija rezultatov bo izvedena na več načinov. Uporabili bomo robustno Javascript knjižnico Benchmark.js[4], ki deluje na večini platform, podpira časovnike visoke resolucije in izmeri statistično zelo natančne rezultate. Uporablja se za performančne regresijske teste na kritičnih delih zahtevnih aplikacij. Omogoča primerjavo zmogljivosti s preteklimi rezultati in jasno pokaže naše napredovanje ali nazadovanje. Temelji na statističnih praksah, kot sta deviacija in varianca.

Natančno bomo analizirali delovanje implementacij z uporabo Chrome DevTools[5]

[3] New video policies for iOS: webkit.org/blog/6784/new-video-policies-for-ios
[4] Knjižnica Benchmark.js: github.com/bestiejs/benchmark.js
[5] Chrome DevTools: developer.chrome.com/devtools

orodja v namiznem brskalniku in mobilnih napravah, kjer je to le mogoče. Raziskali bomo vplive uporabe vzporednih niti na glavno nit in delovanje brskalnika pod obremenitvijo.

Merilo bo število sličic na sekundo (angl. *frames per second, FPS*), količinsko beleženje časa dogodkov pri mnogokratnih poizkusih na vseh napravah, ki se bodo ob koncu poslali na oddaljen strežnik in obdelali s primernim orodjem za analizo.

1.7 Pregled dela

V poglavju 2 opisujemo tehnologije, s katerimi smo imeli posreden ali neposreden stik pri razvoju in meritvah delovanja predvajalnika. V poglavju 3 podrobno opisujemo arhitekturo in delovanje predvajalnika ter različne optimizacije spletnih aplikacij za konsistentno delovanje na različnih napravah. V poglavju 4 opisujemo zaporedno in vzporedno različico razvitega predvajalnika. V poglavju 5 predstavimo meritve vpliva vzporednosti na obremenitev glavne niti ter meritve uporabe vzporednosti v mobilnih brskalnikih. Analiziramo vpliv vzporednosti na obremenitev glavne niti in uporabo vzporednosti v mobilnih brskalnikih. V zadnjem poglavju 6 predstavimo sklepne ugotovitve in nakažemo nadaljnje korake.

2 Tehnologije

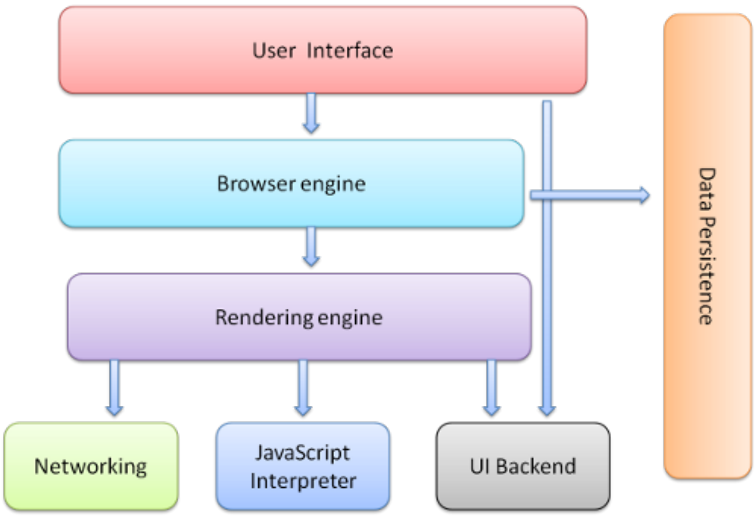
2.1 Spletni brskalniki

V poglavju bomo predstavili spletne tehnologije in platforme, uporabljene pri razvoju video dekodirnika in analizah zmogljivosti njegove zaporedne in vzporedne različice. Opisali bomo visoko-nivojsko delovanje in vzporednost na nivoju brskalnikov ter pregledali delovanje jedra Javascript. Pregledali bomo tehnologije, ki jih lahko uporabimo za dosego vzporednosti na spletu.

Spletni brskalniki so najpogosteje uporabljena programska oprema današnjega časa. Njihova glavna naloga je predstavitev spletnih vsebin, naj bodo to dokumenti HTML, PDF ali kaj drugega. V sodobnih brskalnikih jemljemo hitro izvajanje za samoumevno, a brskalniki sestojijo iz mnogih komponent, od katerih ima lahko vsaka velik vpliv na hitrost izvajanja. Za razvijalce je ključno vsaj približno poznavanje delovanja brskalnikov in njihovih posebnosti, saj slednje pomaga k boljšim odločitvam in razumevanju priporočenih praks.

Glavne komponente brskalnikov so:

- Uporabniški vmesnik (angl. *User Interface*): predstavlja vse, kar na zaslonu vidimo,



Slika 2.1 Glavne komponente brskalnika. (vir: www.html5rocks.com)

z izjemo področja, kjer se vsebina strani prikaže (menijska vrstica, gumb naprej, gumb nazaj, zaznamki, naslovna vrstica ipd.).

- Jedro brskalnika (angl. *Browser Engine*): upravlja akcije med uporabniškim vmesnikom in jedrom za izrisovanje.
- Jedro za izrisovanje (angl. *Rendering Engine*): je zadolženo za izrisovanje vsebine. Različni brskalniki uporabljajo različna jedra za izrisovanje vsebine, najpogosteje WebKit (Google Chrome, Safari).
- Upravljalec omrežja (angl. *Networking*): upravlja omrežne zahtevke z glede na platformo specifičnimi implementacijami, navzven ponuja splošen vmesnik.
- Zaledni sistem uporabniškega vmesnika (angl. *User Interface Backend*): uporabljen za izris osnovnih gradnikov spletnih strani, kot so različni izvlečni meniji, okna ali vnosna polja. Je splošen vmesnik, a v ozadju uporablja gradnike, specifične za operacijski sistem, na katerem deluje.
- Interpreter Javascript (angl. *Javascript Interpreter*): uporabljen za prevajanje in izvedbo prevedene kode jezika Javascript.
- Podatkovna shramba (angl. *Data Persistence*): vztrajnostni nivo, hrani podatke, ki jih brskalniki potrebujejo za daljše časovno obdobje (npr. piškotki).

Potrebno je poudariti, da sodobni brskalniki poganjajo več instanc jedr za izrisovanje, vsak odprt zavihek predstavlja svoje jedro.

V nalogi smo osredotočeni predvsem na najpogostejše brskalnike, ki jih danes najdemo v praksi: Google Chrome in Safari. Osredotočili se bomo na interpreter Javascript in njegov vpliv na jedro brskalnika ter jedro za izrisovanje.

2.2 Jezik Javascript in njegovo jedro

Javascript je dinamični, visoko-nivojski prototipni jezik, ki poleg HTML-ja in CSS-ja predstavlja eno izmed treh glavnih tehnologij spleta. Uporabljen je v vseh spletnih aplikacijah in podprt s strani vseh (modernih) brskalnikov. Podpira tako objektno orientirano programiranje kot imperativno ter funkcijsko. Je interpretiran s strani interpreterja Javascript med izvajanjem. Je široko platformni in nima standardne bitne kode za distribucijo, distribuira se z izvorno kodo. Jezik ni priljubljen le na spletu, ampak se ga vedno več uporablja v aplikacijah, ki tečejo na namiznih in mobilnih aplikacijah. S tem hitrost izvajanja postaja še toliko bolj pomembna.

Ko govorimo o jeziku Javascript, imamo v mislih tudi njegov prevajalnik — ta prevede berljivo izvorno kodo v kodo, ki jo brskalnik lahko izvede. Preprost prevajalnik oz. prevajanje sestoji iz štirih korakov. Prvi korak je semantična preverba (angl. *lexical analyser*), kjer se koda razdeli na posamezne delčke, ki jih imenujemo žetoni (angl. *tokens*). Običajo se za ta korak uporabijo kar regularni izrazi. Žetoni so zatem poslani skozi parser, ki definira strukturo programskih konstruktov in izdela semantično drevo (angl. *syntax tree*). Struktura v obliki grafa se zatem pošlje v prevajalnik, kjer je spremenjena v bitno kodo. Bitna koda se pošlje v bitni interpreter med izvajanjem, kjer je prevedena v nizko-nivojsko kodo in izvedena med delovanjem programa.

2.3 JIT

Klasična arhitektura prevajalnikov je aktualna že vrsto let. Ker so zahteve namiznih aplikacij popolnoma druge kot zahteve spletnih, je klasična arhitektura neprimerna oz. prepočasna za uporabo na spletu. Izboljšavo predstavljajo prevajalniki JIT (angl. *Just-In-Time*). Osnovni problem klasične arhitekture je hitrost prevedbe bitne kode programa v nizko-nivojski jezik med delovanjem. Zmogljivost se lahko izboljša z dodatnim korakom prevajanja bitne kode v nizko-nivojsko — tu opcija prevajanja celotne aplikacijske kode ne

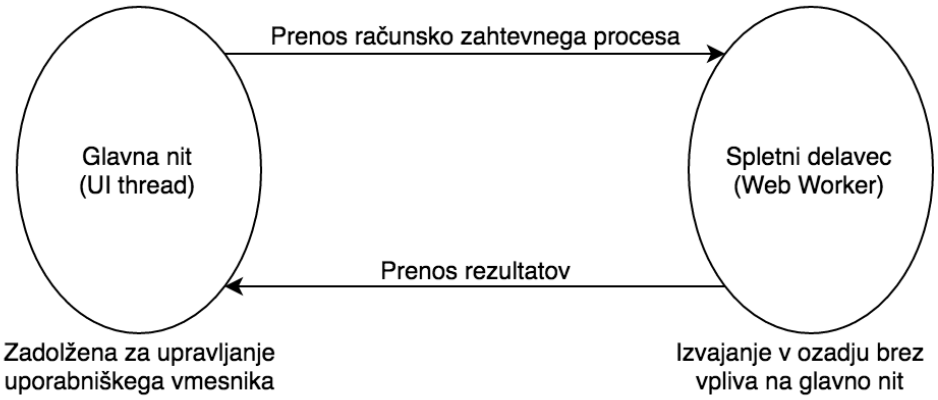
pride v poštev, saj to lahko predstavlja tudi do nekaj minut čakanja. Rešitev (izboljšava) se imenuje leno pravočasno prevajanje (angl. *lazy compilation JIT*). Kot ime nakazuje, prevajalnik prevede bitno kodo v nizko-nivojsko tik preden je to potrebno. Način, kako se prevajalniki tega lotijo, je različen od prevajalnika do prevajalnika. Nekateri optimizirajo funkcije, tretji zanke, četrti iščejo t.i. vroče točke in podobno. Moderno jedro Javascript bo uporabilo večje število prevajalnikov, od katerih bo vsak optimiziral določeno nalogo.

Osnovna optimizacija JIT se izvede le za vroče točke kode, te predstavljajo funkcije, ki se izvedejo vsaj v 100 ponovitvah. Osnovni prevajalnik JIT kliče primerno nizko-nivojsko funkcijo za vsak ukaz bitne kode. Primer delovanja bi bil prvi prevajalnik, imenovan TraceMonkey. Prevajalnik je opazoval potek izvajanja programa in iskal pogosto izvedene dele kode. Te tako imenovane vroče točke je kasneje prevedel v nizko-nivojski jezik. S to optimizacijo je brskalnik Mozilla dosegel 20–40 odstotno izboljšanje zmogljivosti v primerjavi s predhodno različico. Kmalu za tem je Google razvil brskalnik Chrome z jedrom V8, ki je bil zasnovan z namenom hitrega izvajanja. Ta je v celoti preskočil generiranje bitne kode ter je neposredno izvajal nizko-nivojsko kodo. V letu dni od izida jedra V8 je Google razvil nov način rezervacije prostora z uporabo registra, izboljšal hitrost jedra za regularne ukaze za faktor 10 ter z vidika brskalnika pohitril njegovo delovanje za 150 odstotkov. Danes se razvijajo jedra, ki poleg omenjenega uporabljajo še dodaten korak — usmerjen graf poteka, ki ga prevajalnik uporablja za bolj optimalno sklepanje in pohitritve oz. prevedbe najpogosteje izvedenih delov [7].

2.4 Vzporednost v brskalnikih

Jedro Javascript je eno-nitno programsko okolje. Jezik ne podpira vzporednosti, kar pomeni, da več skript istočasno ne mora teči. Ena nit je tako zadolžena za upravljanje dogodkov uporabniškega vmesnika, upravljanje uporabniške interakcije in vmesnika kot celote ter procesiranja velike količine podatkov aplikacijskega programskega vmesnika (angl. *kratica API*). Razvijalci oponašajo vzporedno izvajanje z uporabo določenih konstruktov, kot so `setTimeout()`, `setInterval()`, `XMLHttpRequest`, ter upravljalcev dogodkov (angl. *event handlers*). Ti konstrukti tečejo asinhrono in ne zaklepajo izvajanja glavne niti, a to ne pomeni da tečejo vzporedno. Asinhroni dogodki so izvedeni, ko se izvajanje trenutno izvajajoče funkcije zaključi.

Namesto uporabe takšnih trikov imamo v HTML5 na voljo spletne delavce. Definicija



Slika 2.2 Relacija med nitjo spletnega delavca in glavno nitjo, ki upravlja uporabniški vmesnik.

standarda HTML5 določa aplikacijski programski vmesnik za izvajanje skript v ozadju znotraj spletnih aplikacij. Spletni delavci omogočajo izvajanje dalj časa trajajočih skript v ozadju za izvedbo računsko zahtevnih nalog, brez vpliva in zaklepanja uporabniškega vmesnika (glavne niti) ali drugih, v ozadju delujočih niti. Glavna nit je še vedno edina zmožna upravljati z uporabniškim vmesnikom (glej poglavje 2.2). Z odlaganjem zahtevnih operacij v ozadje lahko močno razbremenimo glavno nit in omogočimo nemoteno in gladko delovanje uporabniškega vmesnika. Rešitev je razvita in podprta na nivoju brskalnika in ne jezika.

Skripte, ki tečejo na ločeni niti, so mnogo bolj omejene kot tiste na glavni. Ni jim dovoljeno upravljati in uporabljati elementov DOM, njihovih operacij, oken, objektov oken in podobno. Enako velja za dostop do podatkov — delavec ne omogoča neposredne izmenjave podatkov z glavno nitjo. Za prenos podatkov se zahtevajo transakcije.

Poznamo dve vrsti spletnih delavcev, *dodeljene delavce* (angl. *dedicated workers*), ki so dostopni le skripti, ki jih kreira in zažene, ter *deljene delavce* (angl. *shared*), do katerih lahko dostopa katerakoli skripta, ki teče na isti domeni. Za naš namen pridejo v poštev le dodeljeni delavci, v nadaljevanju spletni delavci. Spletni delavci tečejo v ločenih nitih. Koda, ki se na njih izvaja, mora biti ločena v svoji datoteki, saj glavna nit, ki delavca kreira, vanj prenese celotno izvorno kodo ob kreiranju asihnrono.

Komunikacija med delavcem in glavno nitjo poteka preko dogodkov in metod za pošiljanje sporočil, ki lahko sprejmejo tako besedilna sporočila kot objekte JSON. V primeru uporabe enega izmed načinov prenosa, imenovanega *deljena sporočila*, se ta kopi-

rajo, glavna nit in delavec pa ne operirata z isto različico podatka, ampak njegovo kopijo. Večina brskalnikov implementira *algoritem kloniranja*, ki dovoli uporabo in prenašanje strukturiranih tipov, kot so `File, Blob, ArrayBuffer` in podobno. V tem primeru se še vedno naredi kopija podatka, katere posledica je lahko velika količina kopiranih podatkov pri delu z nitmi. Kot alternativo lahko uporabljamo *prenosljive objekte* (angl. *Transferable Objects*). Podatki se prenesejo iz ene niti v drugo. Tu se podatki ne kopirajo, kar močno vpliva tako na hitrost kot na porabo pomnilniškega prostora (prenos po referenci). Edina omejitev tega načina je zaklep objekta na niti, s katere se je zgodil prenos. Objekt lahko uporablja le nit, v katero je bil podatek prenesen nazadnje. Delavci lahko poženejo druge delavce, kar je odlično za nadaljnje izvajanje časovno zahtevnih nalog. Za vsakega delavca brskalnik požene nov proces — ta je enakovreden odprtju novega zavihka brskalnika [2].

2.5 Web GL

WebGL je aplikacijski programski vmesnik jezika Javascript za risanje in upravljanje z interaktivno 2D in 3D računalniško grafiko znotraj kateregakoli brskalnika brez uporabe zunanjih vtičnikov. V popolnosti je vgrajen v vse spletne standarde, ki dovolijo grafično pospeševanje (npr. HTML5 Canvas). Elemente WebGL uporabljamo posredno preko različnih elementov HTML (npr. HTML5 Canvas) ali preko različnih opcij jezika CSS. Programi sestojijo iz kontrolne kode, napisane v jeziku Javascript, in kode, ki se izvede na grafični procesni enoti.

2.6 Mpeg1

Mpeg1 je uveljavljen standard za kompresijo videa in zvoka z minimalnimi izgubami kvalitete. Uporablja se za zapis videa na zgoščenke, prenos po digitalnih kablih na TV sprejemnike in širokopasovno oddajanje zvoka. Standard natančno definira obliko zapisa videa in njegovo dekodiranje, ne definira pa, kako naj bo zapis izveden tako, da lahko pride do velikih razlik v učinkovitosti zapisanega videa v odvisnosti od uporabljenega enkodirnika. Podpira velikosti videov do 4095 x 4095 ter bitnih hitrosti do 100 Mbit na sekundo, a so najpogosteje uporabljene velikosti videa 320 x 240 slikovnih točk.

V današnjem času so Mpeg1 standard nadomestili novejši standardi (npr. Mpeg4), a smo omenjen standard v nalogi uporabili predvsem zaradi nižjih zahtev po virih ter lažje

implementacije dekodirnika v jeziku Javascript.

2.7 FFmpeg

Je vodilno multimedijsko programsko ogrodje za enkodiranje videov in zvoka ter programsko obdelavo videa nasploh. Zmožen je obdelovati skoraj vse obstoječe video in zvočne formate.

2.8 Google Chrome DevTools

Je množica orodij za razhroščevanje in razvoj spletnih aplikacij, vgrajenih v brskalnik Google Chrome. Razvijalcem omogoča dostop do podrobnosti delovanja aplikacij v brskalniku. Omogoča izvajanje zmogljivostnih, grafičnih, omrežnih in mnogih drugih analiz ter kot rezultat kakovostno optimizacijo aplikacij, ki v polnosti izkoriščajo delovanje brskalnikov.

3 Delovanje video predvajalnika

3.1 Arhitektura predvajalnika

V poglavju opisujemo strukturo in delovanje video predvajalnika, razvitega za namen meritev in analize izboljšanja zmogljivosti. Razvili smo Javascript video dekodirnik, ki vse delo opravi na glavni niti, in njegovo vzporedno različico. Dekodirnik je samozadosten in sloni le na brskalniku mobilne naprave ter ne uporablja zunanjih virov za pripravo vsebine (npr. odlaganje procesiranja v oblak).

Do danes je bila uporaba dekodirnikov, razvitih v jeziku Javascript, edina možnost za samodejno predvajanje videa na večini mobilnih brskalnikov sodobnih operacijskih sistemov, saj je bilo takšno predvajanje z uporabo sistemskih predvajalnikov nemogoče. Predvsem zaradi povečanega prenosa podatkov pri uporabi videa se je za predvajanje sistemskega videa zahtevala uporabniška interakcija. Zaradi zahtevne implementacije dekodirnikov, premalo zmogljivih naprav in jezika Javascript so bili Javascript dekodirniki v praksi redki. Aktivno so se rešitve problema te vrste pričele uporabljati v času izdelave pričujočega dela, največ zaradi pritiska velikih ponudnikov spletnih vsebin, oglaševalcev in hitro naraščajoče vloge ter pomembnosti videa na mobilnih napravah. Video deko-

dirnik uporabljamo kot breme za dosego bolj nazornega prikaza možnih pohitritev pri uporabi vzporednosti v brskalnikih.

Razvit predvajalnik dekodira video, zapisan v formatu Mpeg1. Pri razvoju modulov predvajalnika, ki skrbijo za dekodiranje videa, smo se oprli na že razvit dekodirnik Dominica Szablewskega[1], v nadaljevanju izhodiščni dekodirnik. V našem dekodirniku smo uporabili nekaj njegovih glavnih funkcij dekodiranja, ki smo jih za uporabo v našem predvajalniku ustrezno prilagodili in izboljšali. Rešili smo večino omejitev izhodiščnega dekodirnika, kot je zahtevan prenos celotnega videa pred pričetkom dekodiranja — video uspemo predvajati že med aktivnim prenašanjem. Prav tako smo uspeli z novim dekodirnikom pridobiti na velikosti izvorne kode (4kB napram 76,5kB).

Nov predvajalnik sestoji iz več modulov (glej sliko 3.1). Za lažjo predstavo aritekture predvajalnika, samega delovanja ter medsebojne interakcije modulov, na katere se pri opisu arhitekture in delovanja predvajalnika sklicujemo, si bomo v nadaljevanju pogledali njihove zadolžitve. Module *AssetLibrary*, *AssetLoader*, *NetStream*, *Accumulator*, *Adapter* oz. *Ticker*, *VideoComponent*, *JsVideoPlayer*, *StateObject*, *EventEmitter*, *Helpers* in *Utility* smo v celoti razvili sami. Modul *PictureDecoder* v celoti temelji na funkcijah izhodiščnega predvajalnika. *ContainerDecoder* in *Renderer* le delno temeljita na izhodiščnem predvajalniku.

AssetLibrary

Upravlja prenos in hrani prenesene video posnetke, ki jih med predvajanjem potrebujemo.

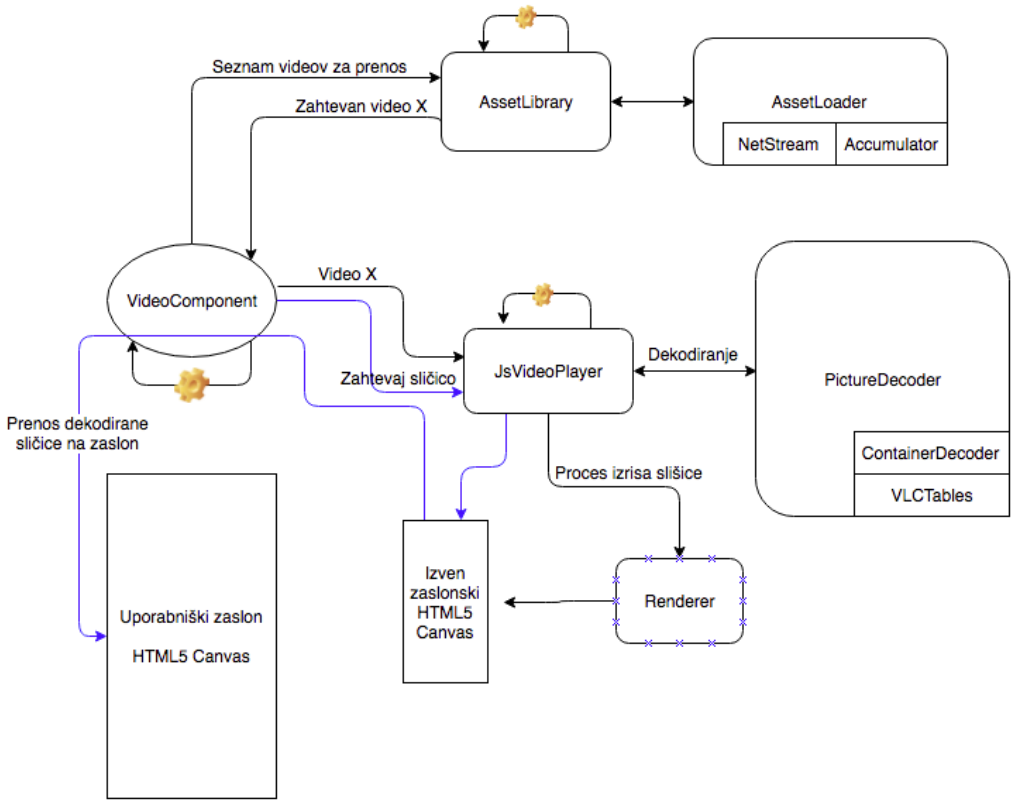
AssetLoader

Omogoča različne načine nalaganja video vsebine — nalaganje v celoti, nalaganje posameznih delčkov videa (angl. *progressive loading*) ali uporabo zunanjih virov (npr. samostojno nalaganje s strani sistemskega predvajalnika).

NetStream

Upravlja omrežne zahtevke.

[1] Dekodirnik: github.com/phoboslab/jsmpeg



Slika 3.1 Nizko-nivojska arhitektura predvajalnika.

Accumulator

Je notranja struktura, ki zna sestaviti in razstaviti po delčkih zahtevano video vsebino v pravilno celoto. Predstavlja osnovni podatkovni tip video vsebine pri nalaganju po delčkih.

Adapter/Ticker

Predstavlja novo razvito funkcionalnost, ki se ukvarja z optimalnim risanjem grafike na zaslon. Uporablja se tudi kot notranja ura naše aplikacije. Na sliki 3.1 prikazan kot rumeni zobnik.

VideoComponent

Glavna komponenta, aplikacijski programski vmesnik predvajalnika. Omogoča zagon predvajalnika. Je vezni člen med uporabniškim vmesnikom in dekodirnikom. Upravlja izrisovanje na zaslon in delovanje predvajalnika; ilustrativno predstavlja glavno nit uporabniškega vmesnika.

JsVideoPlayer

Predstavlja kontrolno komponento video dekodirnika. Z uporabo `PictureDecoderja` dekodira video sličice ter nadzoruje časovno komponento dekodiranja (kdaj mora biti vsebina pripravljena). Upravlja komunikacijo med različnimi komponentami na nivoju dekodiranja videa.

PictureDecoder

Izvaja bitne operacije dekodiranja trenutne slike. Vrne sličico v barvnem zapisu YCbCr[2].

ContainerDecoder

Dekodira AVI paketke, prejete iz omrežja, ter izlušči dolžino in velikost. V izhod posreduje surovo video vsebino, zapisano v formatu Mpeg1.

Renderer

Je zadnji korak pred izrisom na zaslon, predstavlja ovojnico elementa HTML5 Canvas. Izvede pretvorbo iz barvnega prostora YCbCr (izhod dekodirnika) v RGBA prostor, ki ga

[2] YCbCr: ccm.net/contents/753-the-yuv-ycrcb-format

potrebujemo za izris na HTML5 Canvas. Pretvorbe barvnih prostorov so preddefinirane in uporabljene tudi v izhodiščnem predvajalniku (le WebGL del).

VLCTables

Vsebuje identifikatorje različnih okvirjev (angl. *frames*). Tabele so za format Mpeg1 preddefinirane in v enaki obliki uporabljene v izhodiščnem predvajalniku.

StateObject

Omogoča sledenje in upravljanje sprememb podatkov. Preverjamo lahko, ali so bili podatki spremenjeni, jih označimo kot čiste ipd.

EventEmitter

Modul za upravljanje dogodkov znotraj aplikacije. Omogoča proženje in poslušanje dogodkov (jezik Javascript dogodkov kot takih ne pozna, zato smo jih z uporabo preproste podatkovne množice razvili sami).

Helpers

Vsebuje funkcije za delo z objekti (združevanje, kopiranje), funkcije za delo z elementi DOM in vse pomožne funkcije, ki olajšajo izvedbo pogostih nalog.

Utility

Pretvorniki podatkovnih tipov, ki jih potrebujemo v video predvajalniku — pretvorba video vsebine v Uint8Array, detekcija kvalitete zaslona, branje in filtriranje naslovne vrstice brskalnika in podobno.

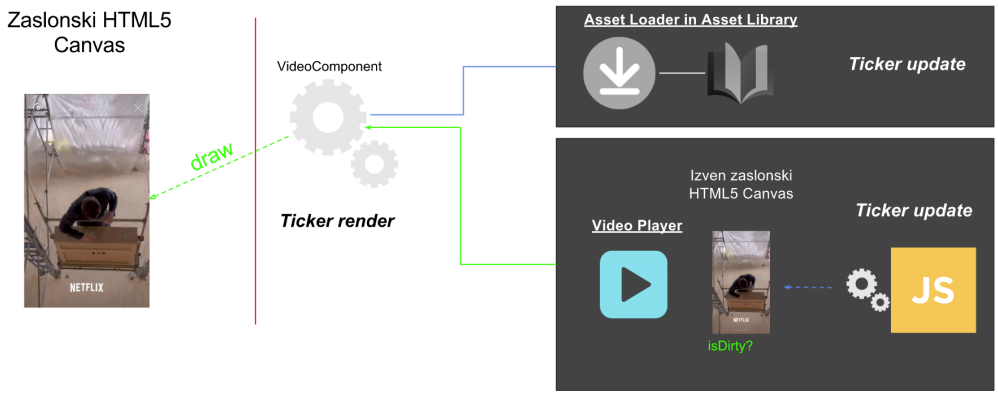
3.2 Notranja ura in izris na zaslon

Modul `Ticker` predstavlja novo razvito funkcionalnost, ki se ukvarja z optimalnim risanjem grafike na zaslon. Posredno ga uporabljamo tudi kot notranjo uro naše aplikacije, od česar zavisi delovanje celotne aplikacije. `Ticker` izrablja s strani brskalnika podprto funkcijo `requestAnimationFrame`. Funkcija je zahtevek za izris grafike, ki ga podamo brskalniku. Izris se zgodi v naslednjem možnem, a nedoločenem trenutku. Brskalnik pri izrisu upošteva vidljivost elementov, velikost elementa, ki se izrisuje in podobno. Grupira vse zahtevke v eno samo osvežitev zaslona (angl. *redraw*). Zagotovi popolno usklajenost

z grafično procesno enoto, cikli osveževanja zaslona in vsemi ostalimi zahtevki za izris ter sprosti centralno procesno enoto.

Z modulom razdelimo časovne rezine, ki jih brskalnik nameni izvajanju naše aplikacije na 2 sklopa, *update* in *render*. Razdelitev reši problem sinhronih risanj na zaslon in optimizira hitrost izrisovanja in delovanja programa, saj zmanjša število preračunavanj, potrebnih za risanje grafike na zaslon. Izogibanje pisanja v DOM znotraj sklopa `update` omogoča ostalim funkcijam tega sklopa branje vrednosti elementov DOM brez vmesnih preračunavanj sprememb, ki se ob pisanju v DOM zgodijo. Združevanje vseh branj v skupine za enkratno izvedbo ter le en izris ob koncu, ki se izvede, ko brskalnik zares osvežuje zaslon, maksimalno optimizira hitrost izrisovanja, običajno brez težav z doseženo hitrostjo 60 sličic na sekundo.

Z uporabo razvitega modula `Ticker` optimiziramo delovanje in izris na zaslon, izboljšamo sledljivost izvajanja in jasnost poteka kode. V sklopu `update` tako vsi moduli izvajajo funkcije, ki so nanj prijavljene — običajno računski del, kot je dekodiranje ali branje iz zaslona. Ko se vse operacije, ki naj bi se izvedle v sklopu `update`, zaključijo, se prične sklop `render`, v katerem rezultate, pridobljene v predhodnem sklopu, izrišemo na zaslon.



Slika 3.2 Upravljanje risanja na zaslon. Razdelitev po sklopih. Vsa vsebina, potrebna za izris sličice videa, se pripravi v sklopu update, v sklopu render pa je vsebina le še izrisana v uporabniški vmesnik (glej Zaslonski HTML5 Canvas) brez dodatnih preračunavanj.

Če delovanje prenesemo na primer našega predvajalnika, izvedemo prenos videa in dekodiranje v sklopu `update`, v sklopu `render` pa na zaslon dekodirano sliko le še izrišemo (slika 3.2). V aplikaciji imamo le eno globalno instanco tega modula, kar omogoča sin-

hrono in sledljivo izvajanje celotne aplikacije.

3.3 Cikel delovanja predvajalnika

Komponenta `VideoComponent` (slika 3.1) predstavlja komunikacijski element, ki povezuje module nalaganja vsebin in dekodirnika videa z uporabniškim vmesnikom. Je vrhnji element predvajalnika, ki ima dostop tako do uporabniškega vmesnika kot shrambe video dekodirnika. Ob kreiranju te komponente preko uporabniškega aplikacijskega vmesnika komponenti podamo element HTML DIV, v katerega želimo izrisovati dekodirano video vsebino. Komponenta kreira element HTML5 Canvas in ga pripne na podan element DIV. Prijavi se na sklopa `update` in `render` modula `Ticker`.

V sklopu `update` komponenta periodično preverja, ali je bilo zahtevano predvajanje videa. V kolikor je trditev resnična, preveri, ali je na voljo video datoteka, ki jo za predvajanje naslednjega videa potrebuje. Če datoteka obstaja, prenese video vsebino k dekodirniku ter požene predvajanje. Od tu naprej v sklopu `update` modula `Ticker` le čaka na konec videa, ko omenjen postopek ponovi za zagon naslednjega videa v vrsti.

V sklopu `render` periodično preverja, ali je stanje modula `JsVideoPlayer` spremenjeno — spremenjene dimenzije ali zapisana nova sličica v izven zaslonski HTML5 Canvas. V kolikor je dekodirnik pripravil novo sličico, jo zapisal v izven zaslonski HTML5 Canvas (glej Izven zaslonski HTML5 Canvas na sliki 3.2), jo modul prepiše na zaslonski HTML5 Canvas, kjer je vidna uporabniku.

Nalaganje video vsebin

Nalaganje video vsebin se v celoti izvede na glavni niti. Ob začetni inicializaciji predvajalnika (VideoComponent na sliki 3.1) se v prvem koraku poženeta modula AssetLibrary in JsVideoPlayer. Modula po vrsti skrbita za upravljanje nalaganja video vsebin in upravljanje dekodirnika. `AssetLibrary` se v prvem koraku prijavi na sklop `update` modula `Ticker` s funkcionalnostjo, ki periodično preverja, ali je datoteka, ki smo jo imeli v prenosu, že prenesena (status prejšnjega prenosa preverjamo z uporabo `StateObjecta`). Če je, prične s prenosom naslednje. Ob zaključku prenosa modul `AssetLibrary` shrani preneseno vsebino skupaj s še nekaterimi meta podatki v razpršeno tabelo, ki omogoča dostop do podatkov v času O(1). Prenosi se izvajajo asinhrono in ne ustavljajo izvajanja glavne niti. Modul `AssetLibrary` ne proži dogodkov. Ko nek zunanji vir potrebuje podatke, jih zahteva od modula, ki pri poizvedbi vrne želeno datoteko ali `null`, v kolikor

ta še ne obstaja oz. ni uspešno prenesena.

Modul `AssetLibrary` potrebuje za izvajanje omrežnih zahtevkov pomožne module `AssetLoader`, `NetStream` in `Accumulator` s slike 3.1 za postopno nalaganje video vsebine, sestavljanje razrezane video vsebine in podobno.

Dekodiranje

Modul `JsVideoPlayer`, ki predstavlja kontrolno komponento dekodirnika, se v prvem koraku prijavi na sklop `update` modula `Ticker` s funkcionalnostjo, ki v vsakem intervalu poizkusi dekodirati eno sličico — v kolikor je to potrebno glede na zapisano hitrost sličic v izvornem videu. `Ticker` izvaja sklop `update` s hitrostjo 60 Hz, kar pomeni, da se v idealnih pogojih izvede 60 klicev na sekundo. V primeru videa, hitrosti 16 sličic na sekundo, se dejansko dekodiranje sličic izvede le 16-krat. Modul izračuna, kdaj mora biti naslednja sličica pripravljena, in jo pred ciljnim časom dekodira z uporabo modula `PictureDecoder`.

Modul `PictureDecoder` predstavlja jedro video dekodirnika, kjer se izvede večina računsko zahtevnih operacij. Dekodira t. i. I-frame, samostojno sličico, ki za dekodiranje ne potrebuje predhodnih sličic. I-frame sličice so JPEG sličice. Dekodira tudi t. i. P-frame, predvideno sličico (angl. *predicted-frame*). S slednjimi se optimizira kompresijo, saj naslednja sličica izhaja iz predhodnje. Ta hrani le spremembe glede na predhodno sličico. Modul vrne sličico v barvnem zapisu YCbCr (luma, rdeča in modra barvna razlika).

Pretvorba in hranjenje dekodirane slike

Po uspešno dekodirani sličici modul `JsVideoPlayer` pošlje sličico v modul `Renderer` (glej sliko 3.1). Modul izvede pretvorbo iz barvnega prostora YCbCr (izhod dekodirnika) v RGBA prostor, ki ga potrebujemo za izris na izvenzaslonski HTML5 Canvas. Kot dodatno optimizacijo izvedemo izris v 3D prostor z uporabo programskega vmesnika WebGL, v kolikor je ta na napravi na voljo (drugače navaden izris v 2D prostor HTML5 Canvasa). Pretvorba barvnega prostora se v primeru uporabe vmesnika WebGL izvede na grafični kartici. Vizualni učinek pretvorbe na grafični kartici je glajenje robov (angl. *antialiasing*), ki vizualno močno izboljša kvaliteto. Uporabili smo obstoječ in preverjen senčilnik (angl. shader) za pretvorbo barvnega prostora formata Mpeg1.

4 Zaporedna in vzporedna implementacija predvajalnika

4.1 Razvoj uporabniško zahtevnih aplikacij

V poglavju opisujemo razlike med zaporedno in vzporedno implementacijo predvajalnika ter predstavimo idejo učinkovite vzporednosti v brskalnikih. Osredotočamo se na težave izvajanja računsko zahtevnih aplikacij v brskalnikih.

V kolikor želimo razviti aplikacijo, ki ponuja visoko odziven uporabniški vmesnik, moramo razumeti delovanje brskalnikov. Ena največjih težav brsklanikov je njihov enonitni model preračunavanja tako za jezik Javascript, kot za upravljanje z uporabniškim vmesnikom. Ko se v brskalniku izvaja koda Javascript, je uporabniški vmesnik neodziven in ne mora sprejemati vhodnih podatkov ali upravljati uporabniške interakcije. Izvajanje kode v polnosti obremeni glavno nit, ki lahko edina skrbi za uporabniški vmesnik. V primeru, da skripta potrebuje več časa za izvedbo ter prekorači omejitev izvajanja brez prekinitve, bo brskalnik uporabniku ponudil možnost prekinitve delujoče skripte (angl. *terminate script*). Te težave se običajno rešuje z razdelitvijo nalog v manjše koščke in uporabo različnih časovnikov, ki jih Javascript ponuja. Med izvajanjem funkcije izvedbo na neki točki prekinemo in naredimo zakasnitev, dolgo nekaj milisekund,

da omogočimo brskalniku izvedbo ostalih čakajočih nalog, povezanih bodisi z upravljanjem uporabniškega vmesnika bodisi česarkoli drugega. Zatem nadaljujemo z izvajanjem predhodne aktivnosti.

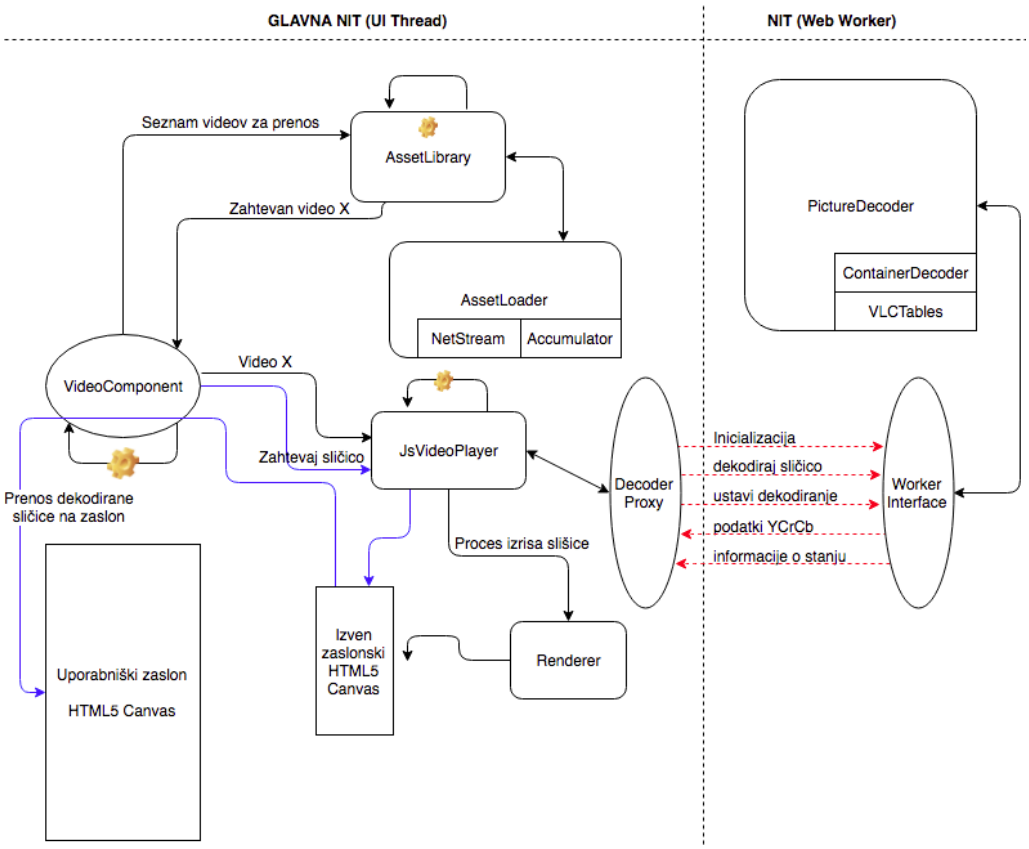
V naši vzporedni različici dekodirnika uporabljamo spletne delavce. Z uporabo delavcev rešimo problem zaklepa glavne niti, ki skrbi za uporabniški vmesnik, saj ustvarimo novo nit, ki v ozadju neprekinjeno izvaja zahtevne, dalj časa trajajoče operacije. V ozir je potrebno vzeti ceno uporabe spletnih delavcev, saj ima uporaba vpliv na zagon aplikacij in porabo pomnilnika, zato moramo pri njihovi uporabi biti prav tako previdni kot pri implementaciji sorodnih eno-nitnih različic. Pri načrtovanju in razvoju spletnih aplikacij je zelo pomembno razumeti tudi razliko med asinhronim izvajanjem (npr. uporaba omrežnih zahtev, ki ne blokirajo izvajanja glavne in dodatne vzporedne niti, saj se izvajajo popolnoma ločeno od našega programa) in sočasno izvedbo z uporabo dodatne vzporedne niti. Slednje predstavlja težavo razvijalcem, ki prihajajo iz drugih jezikov, kot sta Java ali C.

Nekaj primerov operacij, kjer bi uporabili spletne delavce:

- dolgo trajajoče matematične operacije v brskalniku,
- procesiranje velikih podatkovnih zbirk,
- procesiranje multimedijskih vsebin, tako zvoka kot videa,
- delo s slikami ali grafiko na splošno,
- branje in pisanje v lokalne podatkovne shrambe,
- fizika, tj. preračunavanje pozicij, trkov teles in podobno.

4.2 Zaporedna implementacija predvajalnika

V poglavju 3 podrobneje opisan in na sliki 3.1 prikazan predvajalnik predstavlja razvito zaporedno različico. Vsi procesi se izvajajo na glavni niti brskalnika, ki je zaradi zahtevnih in dalj časa trajajočih računskih operacij močno obremenjena. Glavna nit skrbi za celotno procesiranje v aplikaciji, upravljanje z uporabniškim vmesnikom, izrisom na zaslon in izvajanjem akcij v ozadju, klicanih s strani brskalnika (angl. *browser's housekeeping*). Obremenitev aplikacije se pokaže predvsem v neodzivnosti uporabniškega vmesnika med uporabniško interakcijo, počasnostjo procesiranja in morebiti tudi s ponovnim zagonom brskalnika ali ponujeno možnostjo za ustavitev izvajanja skripte.



Slika 4.1 Nizko-nivojska arhitektura predvajalnika pri uporabi spletnih delavcev.

4.3 Vzporedna implementacija predvajalnika

Različico predvajalnika, ki za delovanje uporablja le glavno nit, smo prilagodili do te mere, da za svoje delovanje izkorišča vzporednost v brskalnikih. Moduli predvajalnika ostajajo isti in njihove zadolžitve enake kot pred tem.

V našem predvajalniku smo z meritvami poiskali računsko najbolj obremenilen del in mu dodelili svojo nit (slika 4.1). `PictureDecoder` in ostali moduli, ki skrbijo za dekodiranje videa, predstavljajo po naših meritvah v povprečju 77 odstotkov porabe procesorke moči jedra Javascript (angl. *scripting time*) — v nadaljevanju moduli dekodirnika. Te module smo prilagodili za delovanje na ločeni niti ter jih ločili od ostalih, procesorsko ne tako zahtevnih modulov.

Ker si nit uporabniškega vmesnika in nit, ki poganja module dekodirnika, ne delita pomnilniškega prostora, je bilo potrebno razviti komunikacijski vmesnik, preko katerega

bi lahko kontrolirali module dekodirnika iz kontrolne enote našega video predvajalnika (`JsVideoPlayer`).

Decoder Proxy in Worker Interface

`Decoder Proxy` in `Worker Interface` (slika 4.1) predstavljata komunikacijska modula za komunikacijo med kontrolno enoto video predvajalnika, imenovano `JsVideoPlayer`, in dekodirnikom, imenovanim `PictureDecoder`. Sta programska vmesnika, ki v ozadju namesto neposrednih klicev funkcij pošiljata drug drugemu sporočila. Prvi teče na glavni niti, drugi na niti spletnega delavca.

Dekodirniku `PictureDecoder` ob inicializaciji preko omenjenih komunikacijskih objektov pošljemo video vsebino ter informacijo o merah videa (višina, širina). Dekodirnik, tako kot pri zaporedni implementaciji, dekodira video in kot odgovor vrne dekodirano sličico v formatu YCbCr z uporabo prenosnih objektov (angl. *transferable objects*), omenjenih v poglavju 2.4.

5 Meritve in analiza rezultatov

5.1 Meritve vpliva zaporednega in vzporednega procesiranja na glavno nit brskalnika

Ob pregledu področja smo odkrili mnoge pomanjkljivosti že opravljenih raziskav, pomanjkanje analiz na realnih primerih aplikacij ter veliko neodgovorjenih vprašanj, ki se pojavljajo pri uporabi vzporednosti v spletnih aplikacijah. V poglavju opisujemo meritve in rezultate, ki nam bodo pomagali odgovoriti na vprašanja, zastavljena v poglavju 1.1. V tem podpoglavju obravnavamo vpliv uporabe spletnih delavcev na obremenitev glavne niti, ki skrbi za uporabniški vmesnik, ter prednosti, slabosti in ceno uporabe vzporednosti.

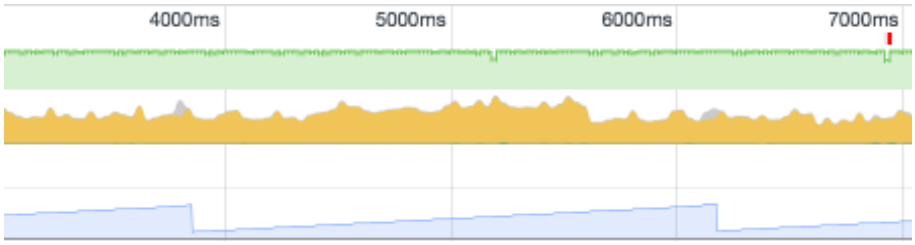
Uporabniki spleta pričakujejo interaktivnost in tekoče izvajanje aplikacij brez zatikanja. Pomikanje po strani mora biti tekoče, takšne morajo biti tudi animacije, ki se na strani izvajajo. Zagotoviti moramo, da se koda, ki jo napišemo, izvaja optimalno. Večina naprav osvežuje zaslon 60-krat na sekundo. Če na zaslonu teče animacija, video ali pa se uporabnik le pomika po strani, mora brskalnik sinhrono z osveževanjem zaslona naprave osvežiti tudi vsebino v brskalniku. Čas, ki ga imamo na voljo na centralno procesni enoti

(CPE), je 16,6 ms. Ker ima brskalnik še drugo delo (angl. *housekeeping*), se razpoložljivi čas zmanjša na vsega 12 ms. Če v okviru tega časa nismo zmožni izvesti potrebnih akcij za izris sličice, je sličica izgubljena, kar se pokaže kot rahla zakasnitev na strani, netekoča animacija ali podobno. Ta pojav s tujko imenujemo 'junk' na spletni strani.

Slika 5.1 Kazalo - vsaka barva označuje tip procesa, ki se je izvajal.

Procesi, ki se v brskalniku izvajajo na CPE, so prikazani na sliki 5.1. Po vrsti predstavljajo:

- Scripting: izvajanje kode v jeziku Javascript.
- Rendering: vse operacije, povezane s stilnimi predlogami (CSS) in preračunavanjem dimenzij elementov DOM.
- Painting: izrisovanje slikovnih točk (angl. *pixels*) na zaslon.
- Other: različne aktivnosti brskalnika, ne nujno povezane z izvajanjem naše aplikacije (angl. *housekeeping*).
- Idle: neaktivnost brskalnika oz. niti odprtega zavihka.



Slika 5.2 Primer izseka časovnice, ki prikazuje delovanje spletne aplikacije. Rdeči odseki nad zelenim grafom označujejo predolge okvirje. Če želimo izrisovati s hitrostjo 60 sličic na sekundo, je lahko okvir dolg največ 16,67 ms, vsak okvir, daljši od te dolžine, je označen rdeče. Meritveno orodje tolerira občasne padce FPS, v kolikor je razlog za (nepojasnjeno) zakasnitev grafična procesna enota (sklepamo iz rezultatov).

Časovnica, pridobljena z orodjem `Google Chrome DevTools`, prikazana na sliki 5.2, prikazuje različne metrike izvajanja v brskalniku:

- število sličic na sekundo (FPS); višji kot so zeleni stolpci, višji je FPS, rdeči odseki nad zelenim grafom pa označujejo predolge okvirje (če želimo izrisovati s hitrostjo

60 sličic na sekundo, je lahko okvir dolg največ 16,67 ms, vsak okvir, daljši od te dolžine, je označen rdeče);

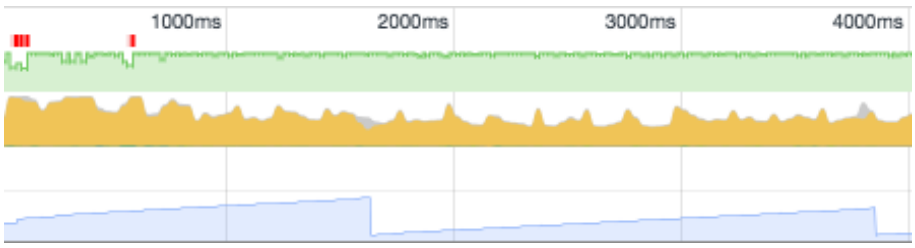
- poraba centralne procesne enote (CPU); na tej sliki večinoma z rumeno barvo narisan graf predstavlja procese, ki enoto uporabljajo; vsak tip procesa je označen z drugo barvo (glej sliko 5.1);
- poraba pomnilnika (HEAP); na sliki predstavljen z modrim grafom.

V prvem sklopu meritev merimo obremenjenost glavne niti pri dekodiranju videa. Obremenitev smo izmerili z uporabo dekodirnika, ki za dekodiranje uporablja le glavno nit, in različice, ki video z uporabo spletnih delavcev dekodira na vzporedni niti. Pri obeh različicah smo izvedli meritve na glavni niti brez in z dodatno obremenitvijo ter s tem simulirali breme, nepovezano z dekodiranjem — tako se približamo bolj realnim rezultatom in lažje opazimo učinke vzporednosti, kot če gre pri vzporednosti le za odlaganje bremena. Opazovali smo vpliv dodatnega bremena na izvajanje. Kot notranji časovnik smo uporabili modul `Ticker`, ki s ferkvenco 60 Hz zahteva novo sličico video dekodirnika. Želimo predvajati 60 sličic na sekundo, kar pomeni, da imamo za vsako sličico 16,67 ms časa. Meritve smo izvedli v časovnem intervalu prvih 10 sekund predvajanja.

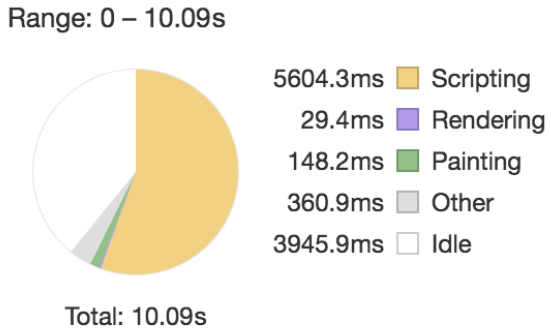
5.1.1 Zaporedno procesiranje brez in z dodatno obremenitvijo

V prvem poizkusu zaporednega procesiranja smo izvedli meritve zmogljivosti dekodirnika, ki za dekodiranje uporablja le glavno nit. Edina obremenitev glavne niti je v tem primeru video dekodirnik. Z občasnimi odstopanji dosegamo ciljni izris 60 sličic na sekundo. Odstopanja vidimo kot rdeče označbe nad zelenim grafom (slika 5.3). Rumeni graf predstavlja izvajanje kode Javascript, tj. celotnega dekodiranja na glavni niti.

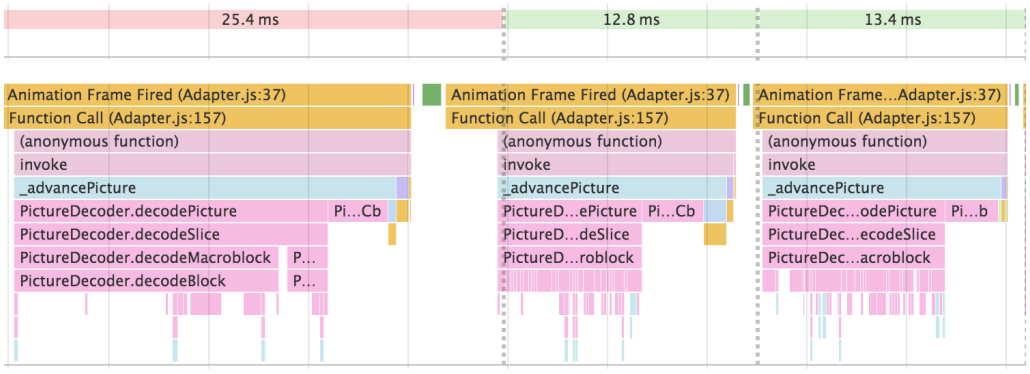
Če se osredotočimo na porabo centralno procesne enote (slika 5.4 in rumeni graf s slike 5.3), se skripta (angl. *Scripting*), tj. dekodiranje in izvajanje kode Javascript, izvaja 55,9 odstotkov časa delovanja aplikacije. Za preračunavanje pozicij elementov v uporabniškem vmesniku (angl. *Rendering*) porabimo 0,3 odstotka celotnega časa. Za risanje vsebine na zaslon (angl. *Painting*) porabimo 1,5 odstotka časa. Brskalnik za svoja opravila potrebuje 3,6 odstotka časa. Neizkoriščen potencial brskalnika (angl. *Idle*) predstavlja kar 39,1 odstotka. Če izvajanja procesov brskalnika in mirovanja ne upoštevamo, potem izvajanje dekodiranja in ostale kode Javascript predstavlja 96,9 odstotkov porabljenega časa.



Slika 5.3 Časovnica dekodiranja videa na glavni niti. Glavna nit izvaja dekodiranje in ni dodatno obremenjena. Zelena barva prikazuje število sličic na sekundo, ki jih med izvajanjem dosegamo. Rumena barva predstavlja porabo procesorske enote s strani kode Javascript. Modra barva prikazuje porabo pomnilnika. Nad zelenim grafom, ki prikazuje število izrisanih sličic na sekundo, vidimo rdeče označbe, ki predstavljajo zakasnitve in padanje števila izrisanih sličic. Ti padci so ob zagonu aplikacij običajni.



Slika 5.4 Poraba centralne procesne enote (CPE) glede na tip procesa pri dekodiranju na glavni niti, ki ni dodatno obremenjena.

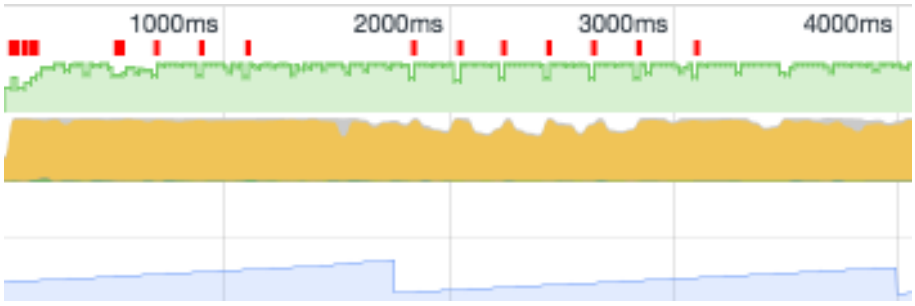


Slika 5.5 Izvajanje aplikacije v omejenih časovnih okvirjih pri dekodiranju na glavni niti. Slika prikazuje prekoračitve okvirja 16,67 ms.

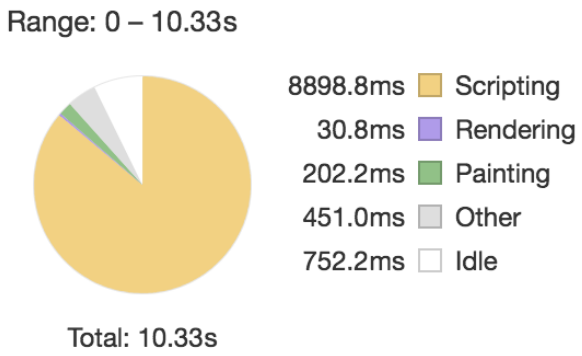
Na sliki 5.5 vidimo primer predolgega izvajanja operacije dekodiranja sličice, ki povzroči zakasnitev izrisa sličice in zamik sličic, ki tej sledijo. Risanje se izvede izven sin-

hronizacije z osvežitvijo zaslona naprave. Večina ostalih okvirjev izvajanja je dolžine, manjše od 16,67 ms.

V drugem poizkusu smo glavno nit, ki dekodira video, dodatno obremenili z izvajanjem nepovezane skripte. Ta v vsakem intervalu, dolgem 16,67 ms, potrebuje 7 ms za izvedbo. Iz slike 5.6 lahko vidimo, da skoraj v celoti obremenimo centralno procesno enoto. Nad zelenim grafom, ki prikazuje število izrisanih sličic na sekundo, vidimo rdeče označbe, ki predstavljajo zakasnitve in padanje števila izrisanih sličic. Teh je precej več kot pri predhodnem poizkusu.



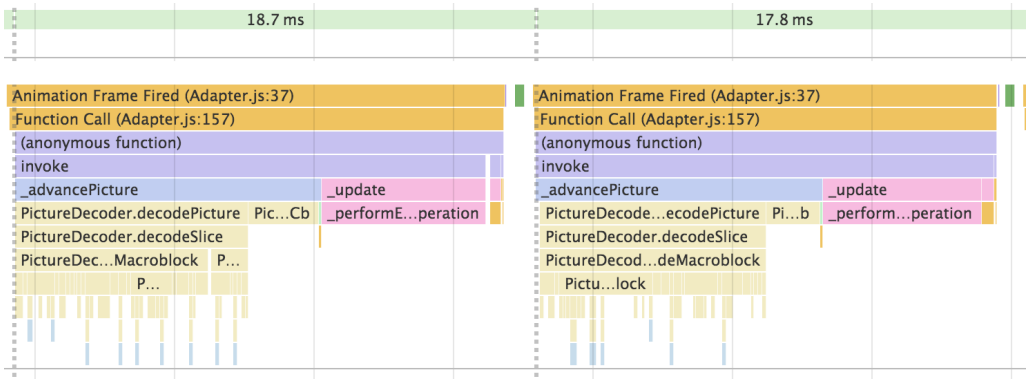
Slika 5.6 Časovnica dekodiranja videa na glavni niti. Glavna nit izvaja dekodiranje in dodatno obremenitev, v povprečju dolgo 7 ms v okvirju dolžine 16,67 ms. Zelena barva prikazuje število sličic na sekundo. Rumena barva predstavlja porabo procesorske enote za izvajanje kode Javascript. Modra barva prikazuje porabo pomnilnika. Rdeče označbe predstavljajo zakasnitve in padce števila izrisanih sličic na sekundo.



Slika 5.7 Poraba centralne procesne enote (CPE) glede na tip procesa pri dekodiranju na glavni niti upoštevajoč vse procese brskalnika. Glavna nit je dodatno obremenjena.

Dekodiranje in izvajanje kode Javascript v tem primeru zaseda kar 86,3 odstotkov časa izvajanja aplikacije (slika 5.7). Za preračunavanje pozicij elementov v uporabniškem vmesniku porabimo 1,9 odstotka celotnega časa. Za risanje vsebine na zaslon potrebujemo

0,22 odstotka časa. Brskalnik za svoja opravila potroši 4,4 odstotka časa. Neizkoriščen potencial brskalnika predstavlja 7,3 odstotka. Hitrost predvajanja niha med 60 in 40 sličicami na sekundo. Roza označen proces s slike 5.8 je nova obremenitev dekodirnika, ki izvajanju doda 7 ms.



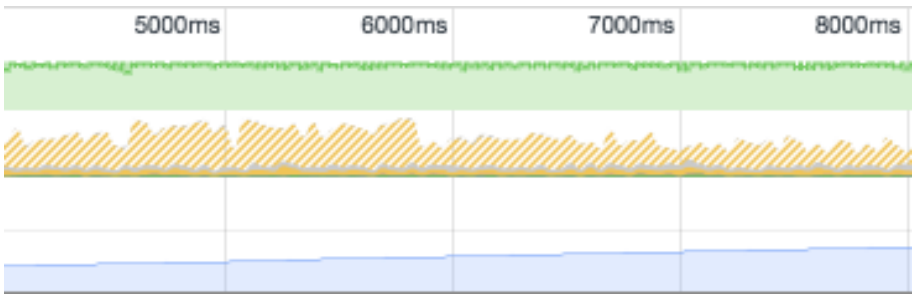
Slika 5.8 Izvajanje aplikacije kot celote v omejenih časovnih okvirjih pri dekodiranju na glavni niti. Slika prikazuje prekoračitve okvirja 16,67 ms. Roza označeni procesi so nova obremenitev dekodirnika.

5.1.2 Vzporedno procesiranje brez in z dodatno obremenitvijo

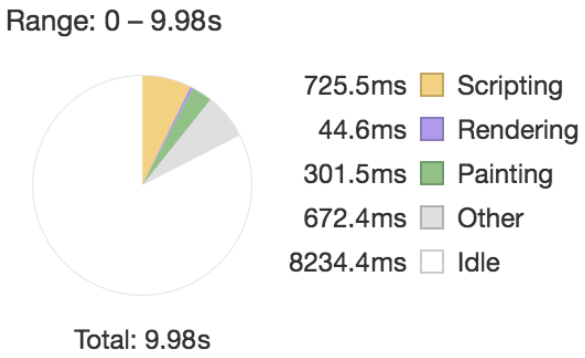
V prvem poizkusu vzporednega procesiranja smo izvedli meritve zmogljivosti dekodirnika, ki za dekodiranje uporablja dodatno nit. Ta skoraj v popolnosti razbremeni glavno nit. Pričakujemo sorazmerno razbremenitev glavne niti glede na odloženo breme, a enako zmogljivost na nivoju celotne aplikacije. Dosegamo ciljni izris 60 sličic na sekundo. Rumeni graf s slike 5.9 predstavlja izvajanje procesov na glavni, šrafiran rumeni graf pod njim pa izvajanje na vzporedni niti. Iz grafa je vidna razbremenitev glavne niti s prenosom bremena na vzporedno nit.

Izvajanje kode Javascript na glavni niti (slika 5.10) predstavlja le 7 odstotkov celotnega časa izvajanja aplikacije. Za preračunavanje pozicij elementov v uporabniškem vmesniku smo porabili 0,5 odstotka celotnega časa. Za risanje vsebine na zaslon potrebujemo 3 odstotke. Brskalnik za svoja opravila potroši skoraj dvakrat toliko kot pred tem, 6,7 odstotka. Neizkoriščen potencial glavne niti brskalnika predstavlja kar 83 odstotkov časa, ki nam je v celoti na voljo.

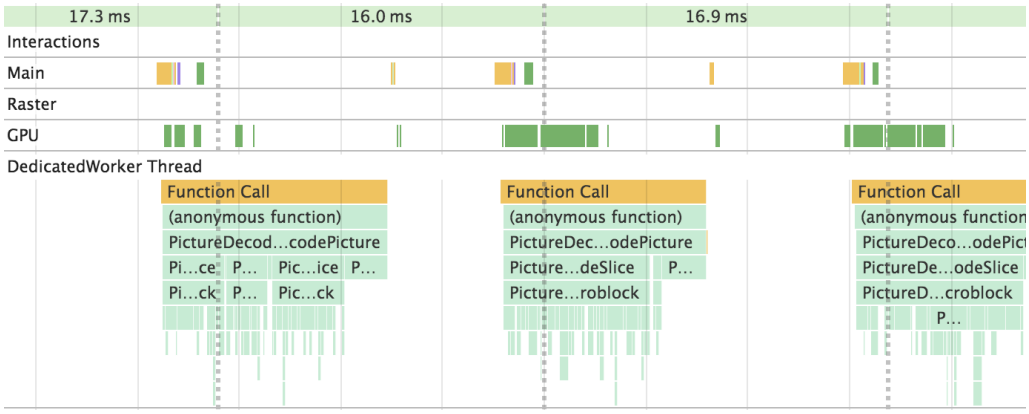
Na sliki 5.11 lahko vidimo razdelitev tipov procesov na glavno in vzporedno nit. Kakšen je posamezen tip procesa, ki se izvaja, razberemo iz barve, glej poglavje 5.1. Prva vrstica pod časovnimi označbami prikazuje čas, porabljen za uporabniško inte-



Slika 5.9 Časovnica dekodiranja videa z uporabo vzporednosti. Zelena barva prikazuje število sličic na sekundo. Rumena barva predstavlja porabo procesorske enote za izvajanje kode Javascript na glavni niti. Šrafirana rumena barva predstavlja izvajanje kode Javascript na vzporedni niti. Modra barva prikazuje porabo pomnilnika.



Slika 5.10 Poraba centralne procesne enote (CPE) s strani glavne niti glede na tip procesa.

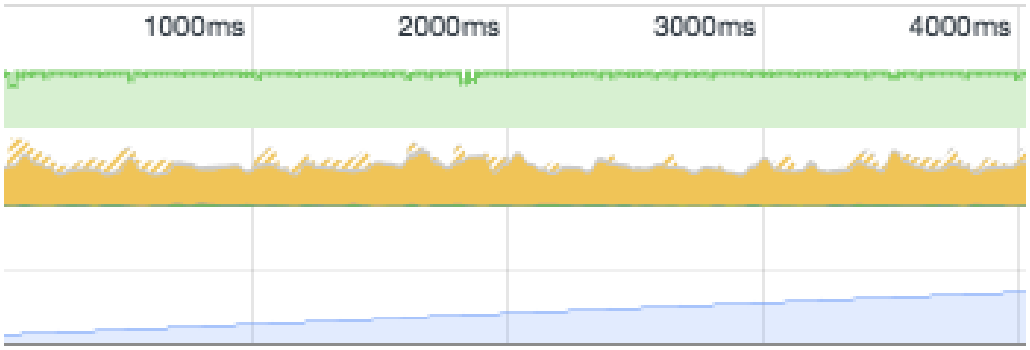


Slika 5.11 Izvajanje aplikacije kot celote v omejenih časovnih okvirih pri dekodiranju z uporabo vzporednosti.

rakcijo (angl. *Interactions*), druga obremenitev glavne niti s procesi, ki so v izvajanju (angl. *Main*), tretja čas, potreben za risanje na zaslon (angl. *Raster*), zelene označbe pa predstavljajo porabo grafične procesne enote (angl. *GPU*). Zadnja vrstica prikazuje

procese na novi vzporedni niti brskalnika (angl. *Dedicated Worker Thread*).

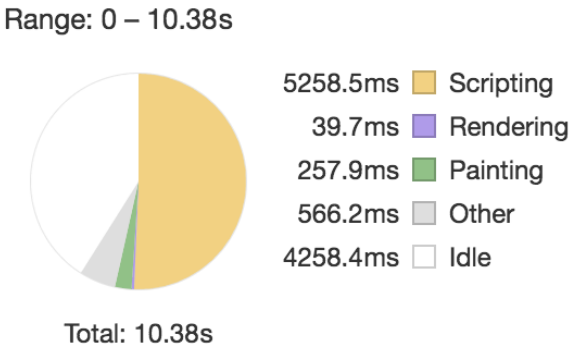
V drugem poizkusu testiranja vzporednega procesiranja smo glavno nit dodatno obremenili z nepovezano skripto, ki v časovnem okviru 16,67 ms potrebuje povprečno 7 ms za izvedbo. Pričakujemo sorazmerno razbremenitev glede na odloženo breme, a izboljšano delovanje na nivoju celotne aplikacije, saj se dodatna obremenitev glavne niti izvaja vzporedno z dekodiranjem na ločeni niti. Dosegamo ciljni izris 60 sličic na sekundo. Poln rumeni graf s slike 5.12 predstavlja izvajanje na glavni, šrafiran rumeni graf pa izvajanje na dodatni niti. Iz grafa lahko razberemo, da smo uspešno razbremenili glavno nit in večino bremena prenesli na vzporedno. Glavna nit ima tako še vedno dovolj časa za izvedbo dodatne obremenitve.



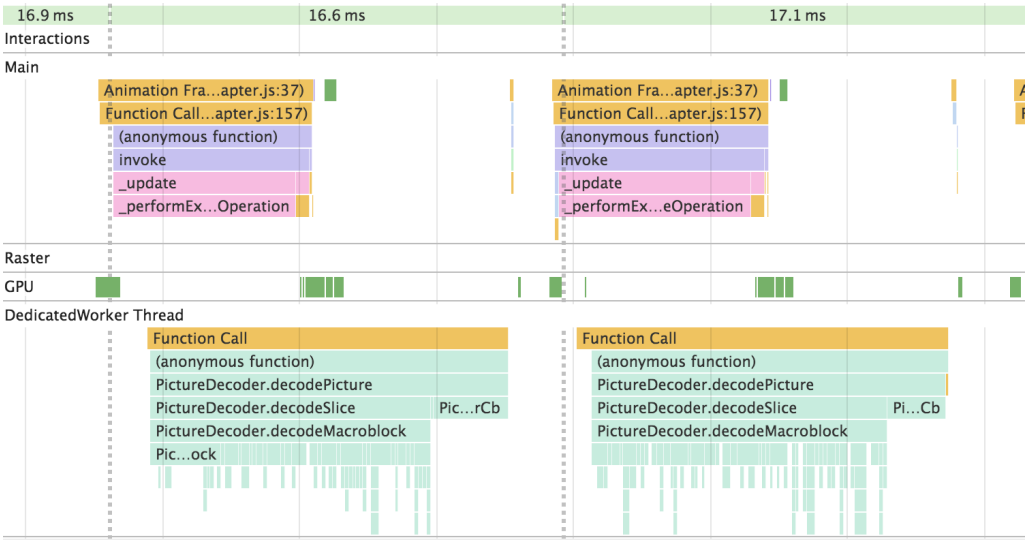
Slika 5.12 Časovnica dekodiranja videa z uporabo vzporednosti. Zelena barva prikazuje število sličic na sekundo. Rumena barva predstavlja porabo procesorske enote za izvajanje kode Javascript na glavni niti. Šrafirana rumena barva predstavlja izvajanje kode Javascript na vzporedni niti. Modra barva prikazuje porabo pomnilnika.

Na sliki (slika 5.13) opazimo močno razbremenitev centralne procesne enote s strani izvajanja kode Javascript. Izvajanje skripte na glavni niti predstavlja 50,7 odstotkov časa, ki nam je na voljo. Za preračunavanje pozicij elementov v uporabniškem vmesniku smo porabili 0,34 odstotka celotnega časa. Za risanje vsebine na zaslon dekodirnik potrebuje 2,3 odstotke časa. Brskalnik za svoja opravila potrebuje 5,5 odstotka. Neizkoriščen potencial glavne niti brskalnika predstavlja kar 41 odstotkov časa, ki nam je na voljo.

Na sliki 5.14 lahko vidimo razdelitev vseh tipov procesov po glavni in vzporedni niti. Kakšen je posamezen tip procesa, ki se izvaja, razberemo iz barve, glej poglavje 5.1. Prva vrstica pod časovnimi označbami prikazuje čas, porabljen za uporabniško interakcijo (angl. *Interactions*), druga obremenitev glavne niti s procesi, ki so v izvajanju (angl. *Main*), tretja čas, potreben za risanje na zaslon (angl. *Raster*), zelene označbe pa predstavljajo porabo grafične procesne enote (angl. *GPU*). Zadnja vrstica prikazuje



Slika 5.13 Poraba centralne procesne enote (CPE) glede na tip procesa pri dekodiranju z uporabo vzporednosti.



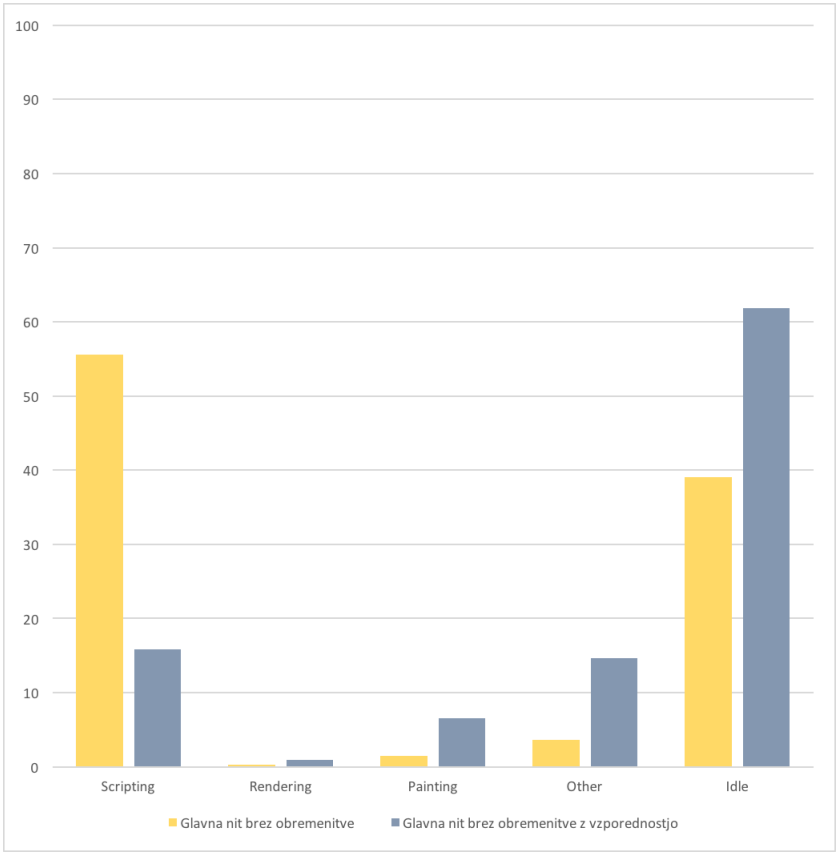
Slika 5.14 Izvajanje aplikacije kot celote v omejenih časovnih okvirih pri dekodiranju z uporabo vzporednosti.

procese na novi vzporedni niti brskalnika (angl. *Dedicated Worker Thread*). Roza označen proces na glavni niti je nova obremenitev dekodirnika, ki izvajanju doda 7 ms.

5.2 Analiza vpliva vzporednosti na glavno nit brskalnika

Pri primerjavi aplikacije, ki za dekodiranje uporablja le glavno nit, ter različice, ki dekodiranje odloži v dodatno nit, je vidna razbremenitev glavne niti (slika 5.15). Obremenitev glavne niti s procesom `Scripting` pade s 55,56 na 15,85 odstotka glede na njeno celotno obremenitev. Procese `Rendering`, `Paiting` in `Other` lahko zaradi zelo spreminjajoče se narave delovanja brskalnikov v našem primeru spregledamo. Nanje vpliva mnogo dejav-

nikov in se lahko od meritve do merive, v neodvisnosti od naše implementacije, precej spreminjajo — vedno predstavljajo najmanj časovno potratne procese. Proces `Other` predstavlja različne operacije brskalnika, tudi upravljanje z dodatnimi nitmi in procesi — ta pri uporabi vzporednosti sicer malenkost naraste, a je razloge za to težko najti, lahko le sklepamo, da dodaten čas predstavlja delo, povezano z upravljanjem vzporednosti. Prosti čas glavne niti `Idle` naraste z 39,11 na 61,89 odstotkov.

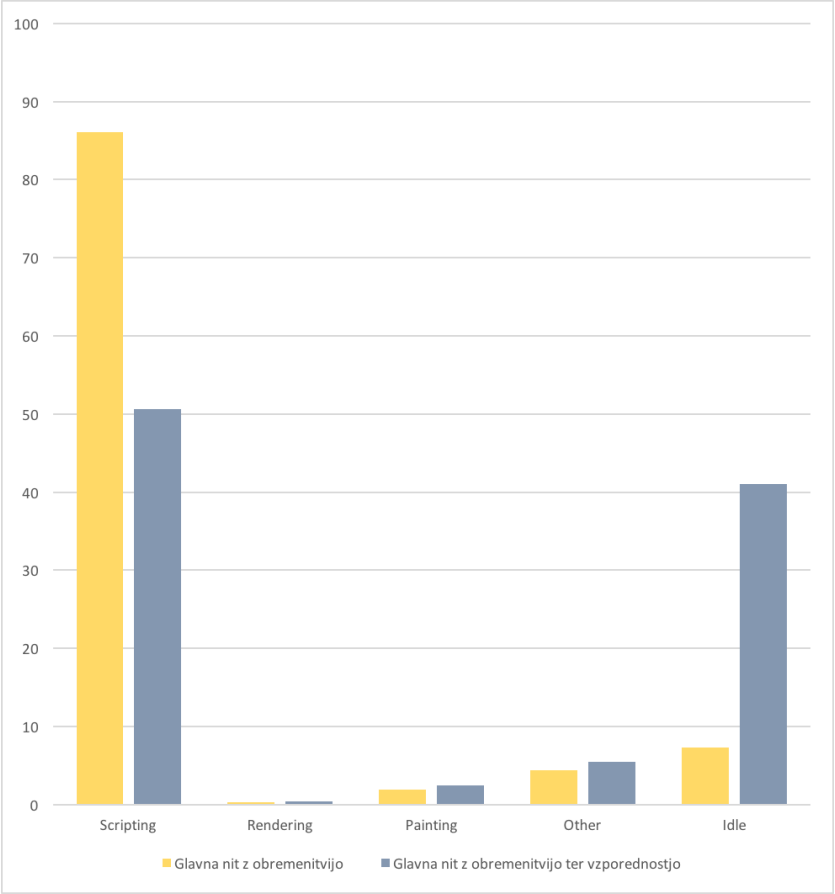


Slika 5.15 Graf na sliki predstavlja porabo CPE v odstotkih s strani različnih procesov, ki v brskalniku tečejo. Prikazana je primerjava petih procesov, ki se v brskalniku izvajajo pri delovanju zaporednega in vzporednega dekodirnika. Glavna nit ni dodatno obremenjena. Temelji na enkratni analizi delujočega dekodirnika v optimalnih pogojih.

Potreben čas za izvajanje kode Javascript na glavni niti z uporabo vzporednosti zmanjšamo za 39,71 odstotkov, a se čas neaktivnosti brskalnika poveča za le 22 odstotkov, kot posledica porasta ostalih procesov brskalnika, ki z nami niso nujno povezani. Glavno nit razbremenimo za 22 odstotkov z 39,11 na 61,89.

Pri primerjavi aplikacije, ki za dekodiranje uporablja glavno nit z dodatno obremeni-

tvijo, ter različice, ki dekodiranje odloži v dodatno nit (dodatna obremenitev ostane na glavni niti), opazimo povečanje izkoristka vzporednosti (slika 5.16).



Slika 5.16 Graf na sliki predstavlja porabo CPE v odstotkih s strani različnih procesov, ki v brskalniku tečejo. Prikazana je primerjava petih procesov, ki se v brskalniku izvajajo pri implementaciji zaporednega in vzporednega dekodirnika. Glavna nit je dodatno obremenjena. Temelji na enkratni analizi delujočega dekodirnika v optimalnih pogojih.

Potreben čas za izvajanje kode Javascript na glavni niti z uporabo vzporednosti zmanjšamo za 35 odstotkov. Procesorski čas, ki ga pridobimo, se poveča za 33 odstotkov s 7,28 na 41,03. Ob veliki obremenitvi glavne niti predstavlja 41,03-odstotna razbremenjenost zelo dober rezultat.

V kolikor celotno breme glavne niti prenesemo na vzporedno nit, gre tu le za odložitev dela in sorazmerno razbremenitev glavne niti v velikosti odloženega bremena. Vzporednega izvajanja v tem primeru ni veliko, se pa poveča delo z upravljanjem komunikacije glavne in vzporedne niti. To lahko na hitrih napravah privede do slabše zmogljivosti

vzporednih različic. Če lahko predvidimo, da je breme, ki ga lahko odložimo, edino večje breme aplikacije ter možnih večjih neodložljivih bremen ni, potem uporaba vzporednosti za razbremenitev glavne niti ni smiselna.

Če je glavna nit dodatno obremenjena in odložimo le del njenega bremena, saj določen del bremena na glavni niti zaradi tehnoloških omejitev ostaja, je uporaba vzporednosti za razbremenitev glavne niti smiselna. V našem primeru odložimo na vzporedno nit le dekodiranje in ohranimo neko konstantno (neodložljivo) breme glavne niti.

Glavno nit razbremenimo za 22 odstotkov, v kolikor odložimo njeno celotno breme na vzporedno nit. V primeru dodatno obremenjene glavne niti pridobimo z uporabo vzporednosti 33 odstotkov procesorskega časa glavne niti. Z naraščanjem neodložljivega bremena glavne niti se izkupiček uporabe vzporednosti veča.

5.3 Meritve zaporednega in vzporednega procesiranja na mobilnih napravah

V tem podpoglavju izvajamo meritve delovanja zaporedne in vzporedne različice dekodirnika z in brez dodatne obremenitve glavne niti na mobilnih napravah. Zanima nas vpliv vzporednosti na delovanje spletnih aplikacij mobilnih naprav v praksi. Z izvedenimi meritvami poizkušamo najti morebitne razlike in odstopanja pri uporabi brskalnikov na različnih operacijskih sistemih in napravah.

V primeru zaporednega izvajanja se je celotno procesiranje dogajalo na glavni niti, v primeru vzporednega pa smo le dekodiranje odložili na dodatno nit. Za predvajanje uporabljamo video dolžine 801 sličice s ciljno hitrostjo predvajanja 60 sličic na sekundo. Merimo hitrost dekodiranja celotnega videa pri največji obremenitvi dekodirnika na 11 najpogostejših mobilnih napravah na trgu ter namiznem računalniku kot refenco. Pri meritvah smo analizirali naprave in operacijske sisteme, prikazane v tabeli 5.1. Na vsaki napravi smo posamezno meritev zaporedne in vzporedne različice ponovili 20-krat. Čas dekodiranja smo izmerili v milisekundah in ga pretvorili v sličice na časovno enoto. Preračunano je meja za dosego izrisa 60 sličic na sekundo pri 13350 ms za dekodiranje videa naše dolžine.

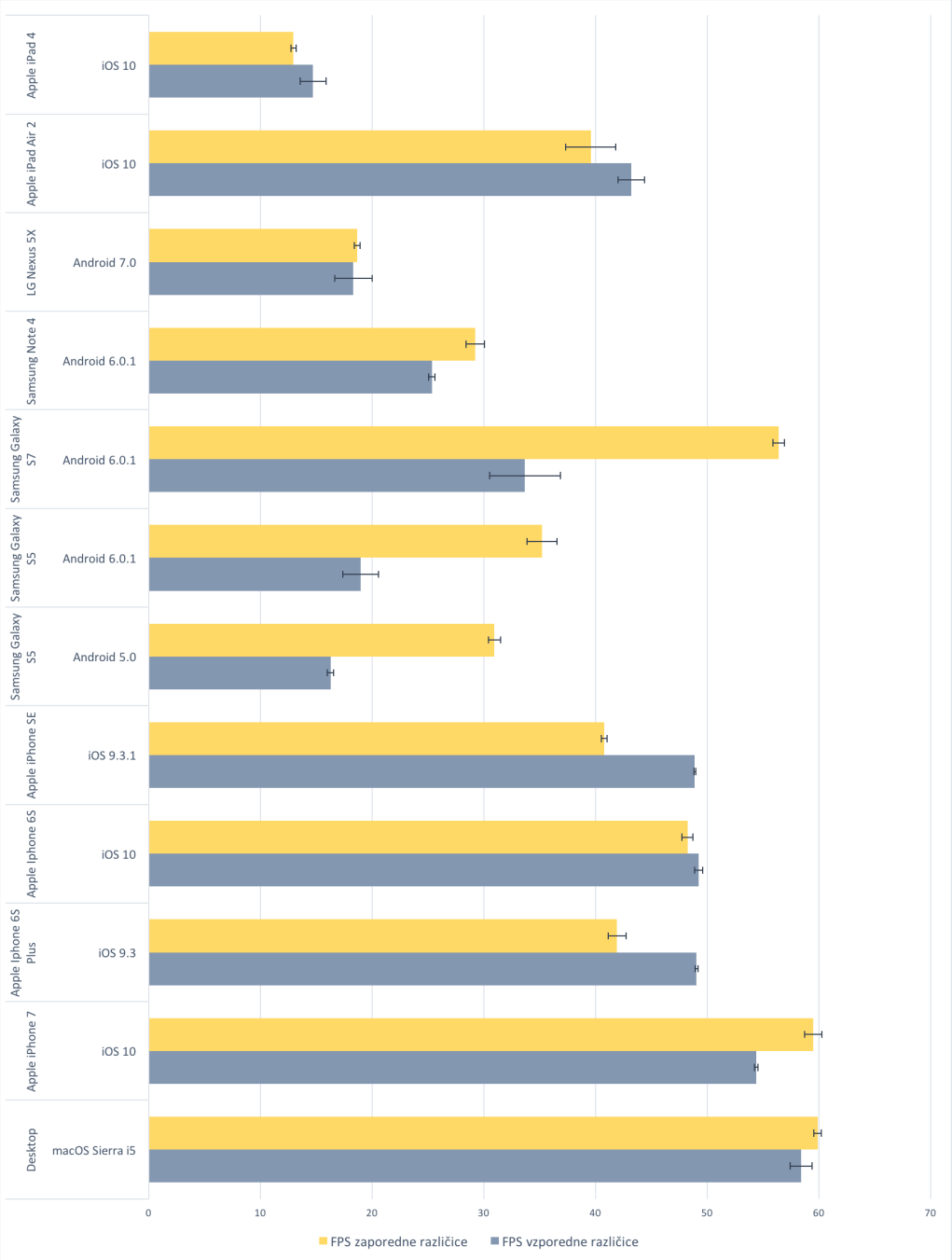
Tabela 5.1 Naprave, uporabljene pri testiranju računske zmogljivosti vzporedne in zaporedne različice predvajalnika.

Naprava	Operacijski sistem	Brskalnik
Desktop	macOS Sierra i5	Chrome
Apple iPhone 7	iOS 10	Safari
Apple Iphone 6S Plus	iOS 9.3	Safari
Apple Iphone 6S	iOS 10	Safari
Apple iPhone SE	iOS 9.3.1	Safari
Samsung Galaxy S5	Android 5.0	Chrome
Samsung Galaxy S5	Android 6.0.1	Chrome
Samsung Galaxy S7	Android 6.0.1	Chrome
Samsung Note 4	Android 6.0.1	Chrome
LG Nexus 5X	Android 7.0	Chrome
Apple iPad Air 2	iOS 10	Safari
Apple iPad 4	iOS 10	Safari

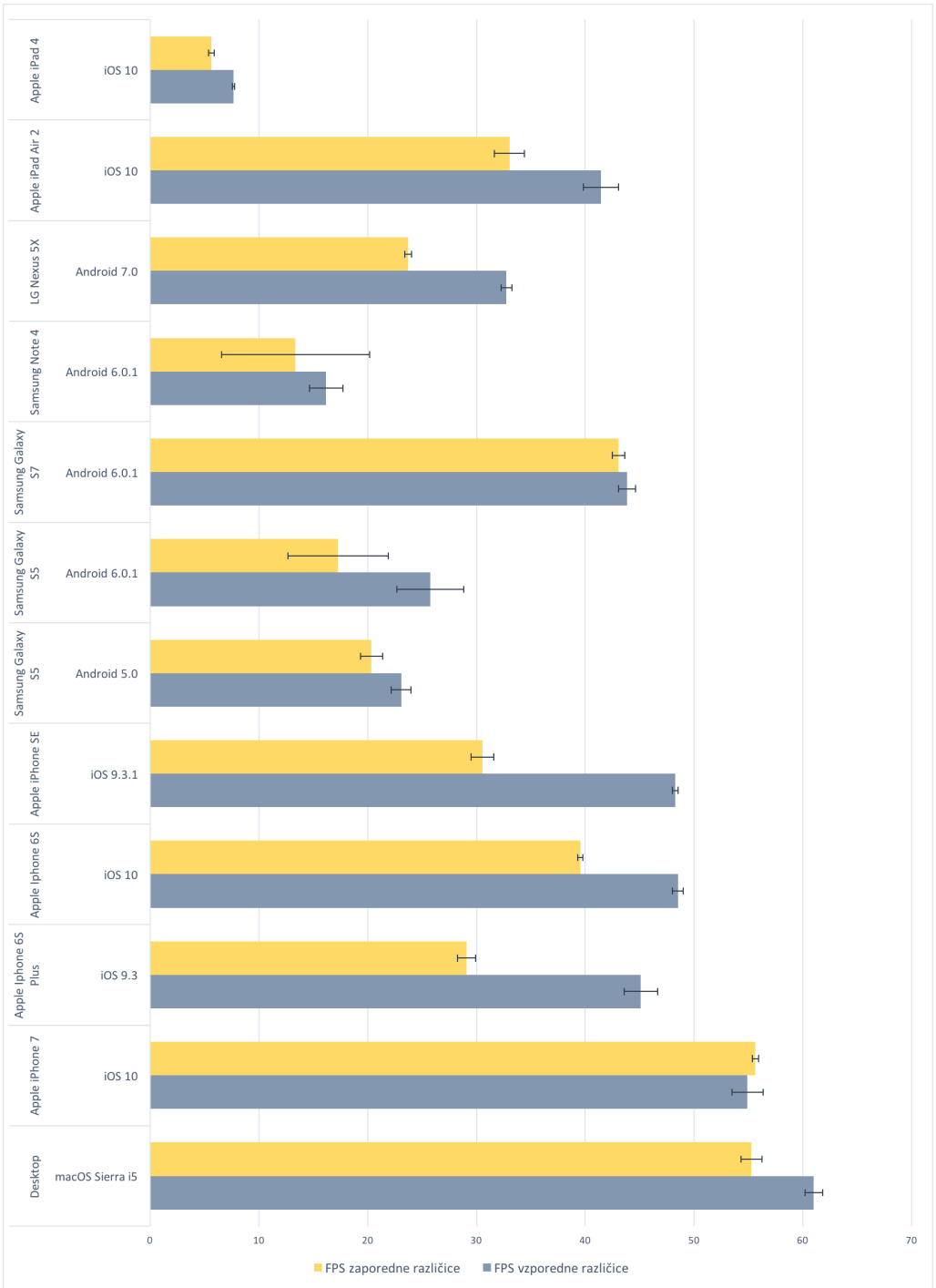
5.3.1 Procesiranje brez in z vzporednostjo ter brez dodatne obremenitve

Izmerili smo čas dekodiranja na zaporedni in vzporedni različici predvajalnika. Glavna nit pri teh meritvah ni bila dodatno obremenjena. Pri obeh različicah smo pričakovali približno enake rezultate v številu sličic na sekundo oz. rezultate malenkost v prid zaporedni različici, ki nima dodatnega dela z upravljanjem vzporedne niti.

Iz slike 5.17 razberemo povprečno število dekodiranih sličic na sekundo s 95-odstotnim intervalom zaupanja. Rezultati so nas presenetili, saj so bila odstopanja v prid zaporedni različici predvajalnika mnogo večja kot pričakovano. Brskalnik namiznega računalnika (na sliki angl. *Desktop*) je dovolj zmogljiv in pri zaporedni implementaciji zlahka dekodira 60 sličic na sekundo, tu z uporabo vzporednosti malenkostno izgubimo. Apple iPhone 7, Samsung Galaxy S5 in Galaxy S7 pri zaporednem dekodiranju dosegajo opazno večje hitrosti dekodiranja kot pri uporabi vzporednosti. Vzporedna različica občutno obremeni predvsem Samsung Galaxy S5 in novejšo različico Galaxy S7, ki ji pri uporabi vzporednosti število sličic pade s 56 na 34 na sekundo. Pri starejših modelih naprav proizvajalca Apple vzporedno izvajanje pohitri dekodiranje za 1–8 odstotkov, kar je sicer zanemarljivo, a presenetljivo, saj pohitritve zgolj zaradi odložitve bremena z glavne niti ne bi pričakovali.



Slika 5.17 Povprečno število izrisanih sličic na sekundo pri največjem izkoristku naprave (v FPS). Prikazuje hitrost zaporednega in vzporednega dekodirnika s 95-odstotnim intervalom zaupanja. Glavna nit ni dodatno obremenjena. Želimo čim večji FPS.



Slika 5.18 Povprečno število izrisanih sličic na sekundo pri največjem izkoristku naprave (v FPS). Prikazuje hitrost zaporednega in vzporednega dekodirnika z dodatno obremenitvijo glavne niti s 95-odstotnim intervalom zaupanja. Želimo čim večji FPS.

5.3.2 Procesiranje brez in z vzporednostjo ter z dodatno obremenitvijo

Izmerili smo čas dekodiranja na zaporedni in vzporedni različici predvajalnika. Glavna nit je bila pri merjenju dodatno obremenjena s procesom, ki v povprečju porabi 7 ms v 16,67 ms dolgem časovnem intervalu. Ker je glavna nit dodatno obremenjena, smo pričakovali pohitritve vzporedne različice.

Iz slike 5.18 razberemo povprečno število dekodiranih sličic na sekundo s 95-odstotnim intervalom zaupanja. Pohitritve vzporednega izvajanja opazimo skoraj na vseh napravah. Opazno je stalno izboljšanje števila dekodiranih sličic na sekundo. Izjema je Apple iPhone 7, ki je že pri samostojni uporabi glavne niti občutno hitrejši od ostalih konkurentov. Pohitritve segajo od 1 do 17 sličic na sekundo oz. 2 do 37 odstotkov v prid uporabe vzporednosti.

5.4 Analiza vpliva vzporednosti na izvajanje mobilnih spletnih aplikacij

Na izvajanje spletne aplikacije lahko vpliva tip brskalnika, v katerem aplikacija teče, operacijski sistem ali operacijski sistem v relaciji z vrsto in znamko naprave. Obnašanje iste aplikacije je lahko v realnosti iz meritve v meritev drugačno. Teoretične predpostavke in izračuni v realnem okolju velikokrat ne držijo ali pa imajo velika odstopanja in mnogo izjem.

V postopku meritev zaporednega dekodiranja smo v prvem koraku glavno nit obremenili le z dekodirnikom, v drugem pa poleg dekodirnika poganjali še dodatno nepovezano skripto. Na sliki 5.19 vidimo, da glavna nit, katere naloga je dekodiranje videa, dosega bistveno boljše rezultate kot glavna nit, ki je poleg dekodiranja dodatno obremenjena. Zmogljivost naprav je različna, od namiznega računalnika in naprave Apple iPhone 7 kot najboljših do naprav Samsung Galaxy S5 in Apple iPad, ki sta kljub novejšemu datumu po zmogljivosti že močno v zaostanku. V kolikor je glavna nit dodatno obremenjena, število dekodiranih sličic na sekundo sorazmerno pada, saj v časovnem okviru 16,67 ms dekodirnik zaradi dodatnih obremenitev ne uspe pripraviti nove sličice za izris. Vse naprave na testu so bile obremenjene do skrajnosti. Predvajanje videa je pričakovano počasnejše pri dodatno obremenjeni glavni niti.

V postopku meritev vzporednega dekodiranja smo v prvem koraku glavno nit popolnoma razbremenili, v drugem pa na glavni niti istočasno izvajali dodatno nepovezano

skripto. Pri tej različici predvajalnika se dekodiranje izvaja na ločeni niti. Število sličic (na sliki 5.20), ki jih dekodnirnik z neobremenjeno glavno nitjo dekodira, je na namiznem računalniku in večini naprav Apple pričakovano večje kot pri različici z obremenjeno glavno nitjo. Konsistentno smo na vseh napravah Android beležili ravno nasproten učinek. V primeru dodatno obremenjene glavne niti število sličic na sekundo naraste oz. drugače, različica, ki ima glavno nit prosto, beleži padec števila dekodiranih sličic v sekundi. Obnašanje je konsistentno pri vseh napravah z operacijskim sistemom Android. Razlika v hitrosti je na novejših napravah izrazitejša.

Pri merjenju zmogljivosti vzporednih implementacij na mobilnih napravah smo zasledili mnogo odstopanj. V primeru uporabe vzporednega dekodirnika in neobremenjene glavne niti smo pričakovali zmogljivost, podobno tisti pri samostojni uporabi glavne niti, ki video dekodira sama — breme dekodiranja smo v celoti odložili na vzporedno nit, tako da poslabšanja ali izboljšanja ne bi pričakovali. Trditev se je izkazala za neresnično. Trditev, da je vzporednost boljša izbira, ko je glavna nit zelo obremenjena in odložimo na nit del bremena, se je izkazala za resnično.

Platforma iOS

Naprave Apple oz. brskalniki, delujoči na operacijskem sistemu iOS, delujejo konsistentno. Bolj kot brskalnik obremenimo, dlje časa potrebuje za izvedbo zahtevanih operacij. Vzporednost v večini primerov nima negativnega učinka, možne so rahle zakasnitve pri uporabi vzporednosti na hitrih napravah ter ob majhnih obremenitvah glavne niti, predvsem zaradi dodatnega dela z upravljanjem niti. Če je glavna nit močno obremenjena in jo razbremenimo z odlaganjem dela bremena na vzporedno nit, uspešno pridobimo 1–37 odstotkov hitrosti (prikazano na sliki 5.21). To se sklada tudi z našimi ugotovitvami v poglavju 5.2, kjer ocenjujemo pohitritev z vzporednostjo pri obremenjeni niti na 33 odstotkov. Meritve na različnih napravah odražajo le rahla odstopanja od teh vrednosti.

Platforma Android

Obnašanje na napravah Android je zelo nepredvidljivo, slika 5.20. Tu pride vzporednost do izraza le, ko na glavni niti istočasno izvajamo druge operacije z znatno porabo procesorskega časa. V kolikor breme glavne niti le odložimo na vzporedno nit, hitrost predvajanja pade. Sprememba v hitrosti je dobro vidna pri napravah Samsung Galaxy S7 in Nexus 5X na sliki 5.20. Sistem Android uporablja razporejevalec procesov, ki se

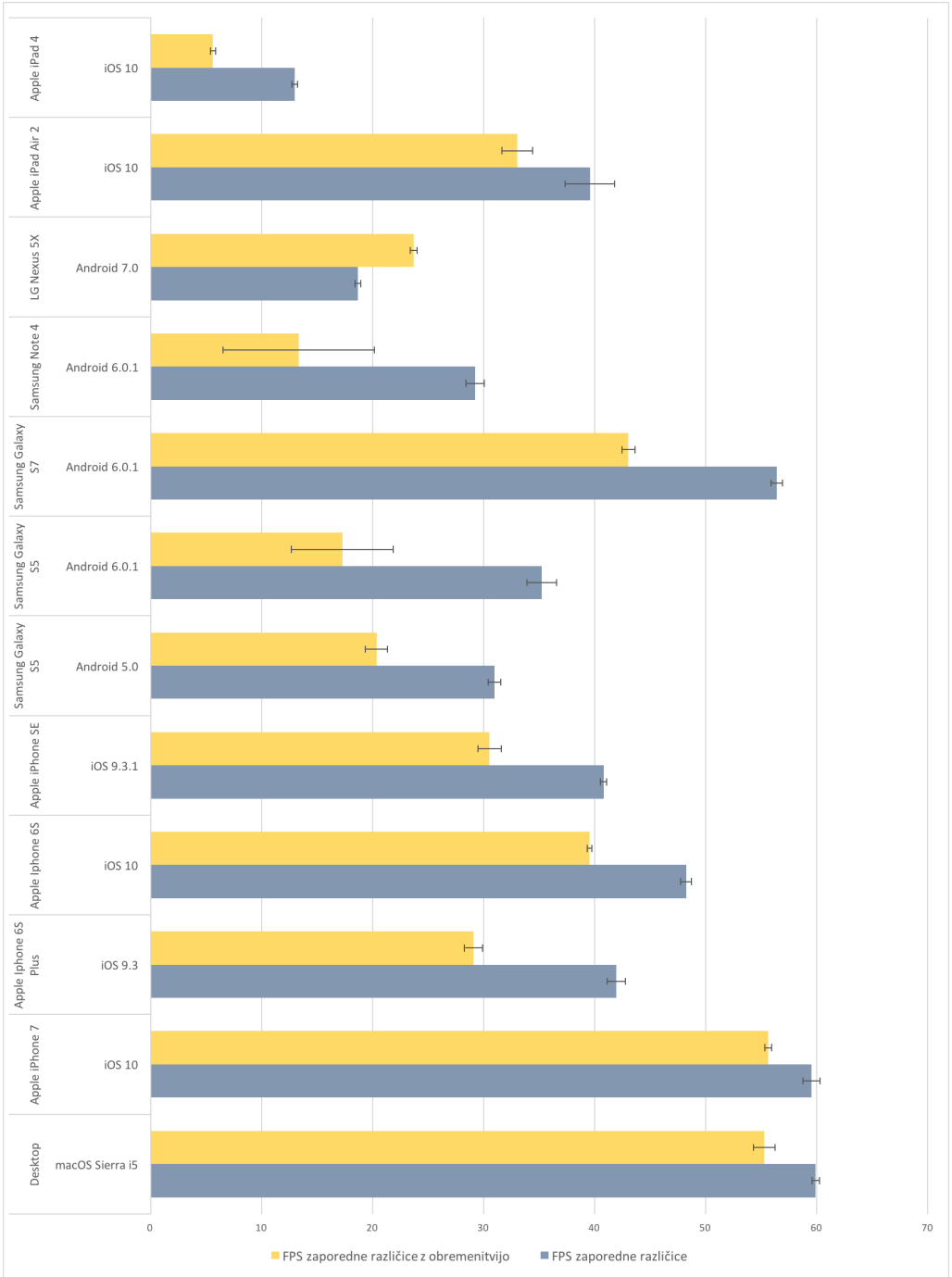
odloča, na kateri procesni enoti bo proces deloval. Iz tu izhajamo, da brskalnik v primeru neaktivne ali malo aktivne glavne niti odloži izvajanje spletne aplikacije na manj zmogljivo centralno procesno enoto. Sklepamo, da izbira procesne enote temelji na aktivnosti glavne niti, ki predstavlja starševsko nit vzporedni niti, ki dekodira video ali izvaja kakršenkoli drug proces. Sklepamo lahko tudi, da brskalnik v procesu optimizacije programa oceni nizek nivo vzporednosti programa in nova nit dejansko ni potrebna, brskalnik tako izvaja le simulacijo.

Primer proti uporabi vzporednosti

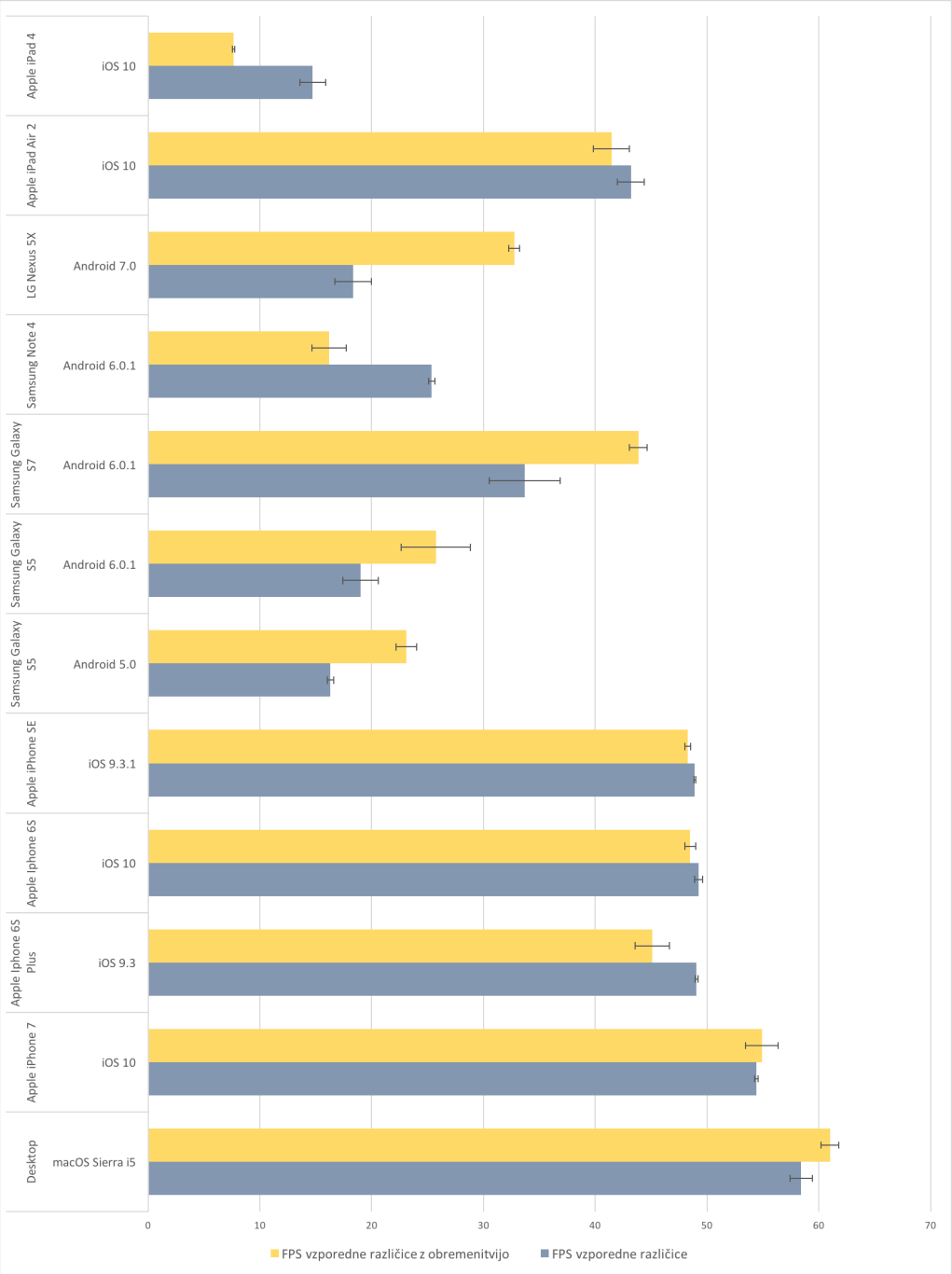
V kolikor celotno breme glavne niti prenesemo na vzporedno nit, gre tu le za odložitev dela in sorazmerno razbremenitev glavne niti v velikosti odloženega bremena. Vzporednega izvajanja v tem primeru ni veliko, se pa poveča delo z upravljanjem komunikacije glavne in vzporedne niti. Če lahko predvidimo, da je breme, ki ga lahko odložimo, edino večje breme aplikacije ter možnih večjih neodložljivih bremen ni, potem uporaba vzporednosti za razbremenitev glavne niti ni smiselna kljub stalni pohitritvi naprav z operacijskim sistemov iOS. Pričakujemo, da se bo s hitrostjo novih naprav trend obračal v prid zaporednega izvajanja. Platforma Android tu močno izstopa z upočasnjenim izvajanjem ob razbremenjeni glavni niti.

Primer za uporabo vzporednosti

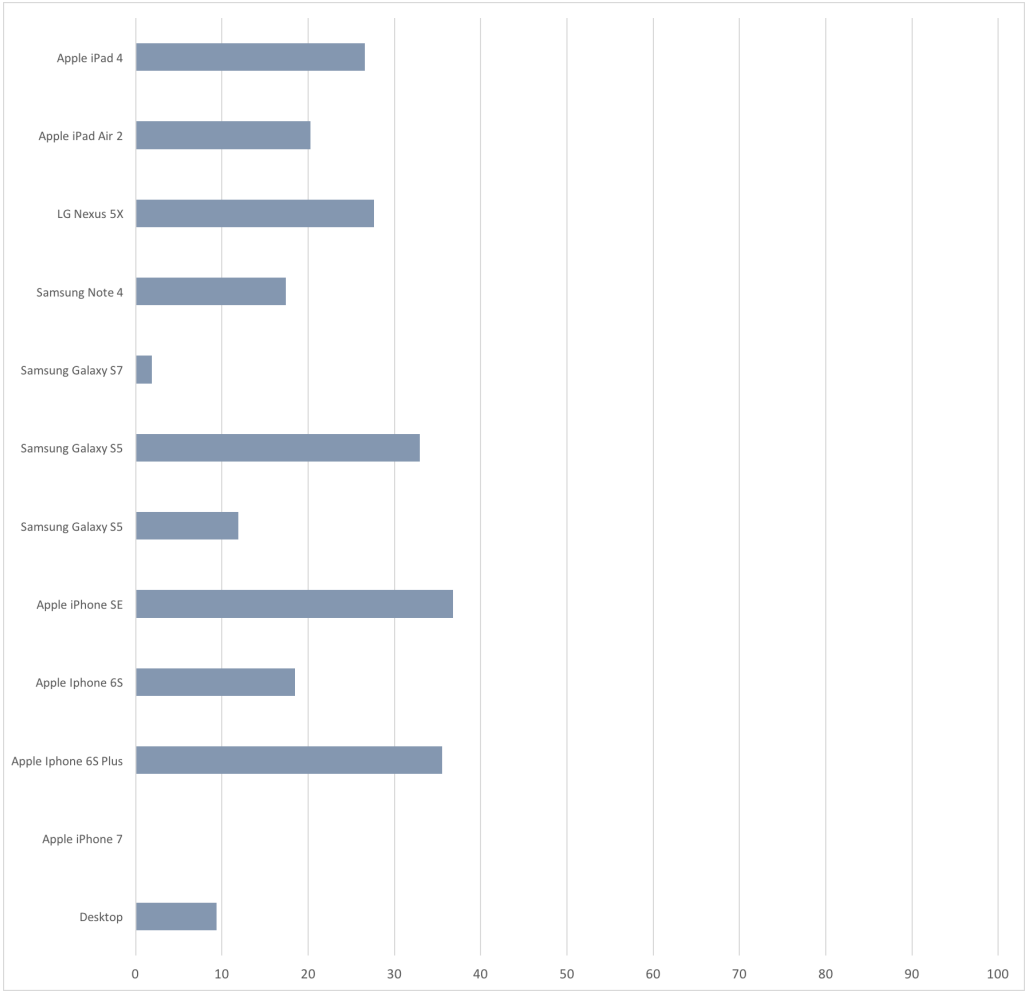
Če je glavna nit dodatno obremenjena in odložimo le del njenega bremena, ker določen del bremena na glavni niti zaradi tehnoloških omejitev ostaja, je uporaba vzporednosti za razbremenitev glavne niti smiselna, tako na platformi iOS kot Android. Napredek znaša med 1–37 odstotka (slika 5.21).



Slika 5.19 Število sličic na sekundo pri zaporedni implementaciji z in brez dodatne obremenitve glavne niti (v FPS). Slika prikazuje zmogljivost zaporednega dekodiranja na glavni niti s 95-odstotnim intervalom zaupanja. Glavno nit v enem primeru obremenimo le z dekodiranjem, v drugem pa z dodatno nepovezano skripto, ki v povprečju za izvedbo potrebuje 7 ms v 16,67 ms velikem časovnem okviru. Želimo čim večji FPS.



Slika 5.20 Število sličic na sekundo pri vzporedni implementaciji z in brez dodatne obremenitve glavne niti (v FPS). Slika prikazuje zmogljivost vzporednega dekodiranja s 95-odstotnim intervalom zaupanja. Dekodiranje izvajamo na dodatni vzporedni niti. V enem primeru je glavna nit popolnoma razbremenjena, v drugem pa dodatno obremenjena s procesom, ki v povprečju za izvedbo potrebuje 7 ms v 16,67 ms velikem časovnem okviru. Želimo čim večji FPS.



Slika 5.21 Pohitritev dekodiranja pri uporabi vzporedne različice glede na zaporedno v odstotkih. Izmerjen napredek po napravi. Izmerjeno pri dodatno obremenjeni glavni niti.

6 Zaključek

Cilj magistrskega dela je bila analiza vpliva vzporednosti na razbremenitev glavne niti in analiza učinkovitosti vzporednosti v mobilnih spletnih brskalnikih različnih naprav pri dekodiranju in izrisovanju videa. Iskali smo odgovor na vprašanje smiselnosti uporabe vzporednosti, ko ta ni implementirana na nivoju jezika. Razvili smo dva predvajalnika, ki za dekodiranje uporabljata:

- glavno nit ali
- vzporedno nit.

Glavna nit brskalnika upravlja z uporabniškim vmesnikom. Deli aplikacij, ki ga upravljajo, se ne morejo izvajati vzporedno. Poleg aplikacije v izvajanju se na glavni niti izvajajo tudi ostali procesi brskalnika. Njena preobremenjenost ima tako lahko velik vpliv na odzivnost in delovanje uporabniškega vmesnika. Na brskalniku osebnega računalnika smo z uporabo vzporednosti poizkušali razbremeniti glavno nit. Pri uporabi dodatne vzporedne niti smo sprostili do 22 odstotkov procesorskega časa glavne niti. Če smo na glavni niti ohranili del bremena, smo v povprečju sprostili 33 odstotkov procesorskega

časa glavne niti napram različicam, ki vzporednosti ne izkoriščajo. Do podobnih ugotovitev smo prišli tudi na napravah, kjer smo pohitritve izmerili v številu izrisanih sličic na sekundo. Z uporabo vzporednosti smo na napravah iOS zabeležili pohitritev od 1 do 37 odstotkov v hitrosti risanja na zaslon. Pohitritev je bila prisotna na vseh napravah iOS. To ne velja za naprave Android, saj je bilo dekodiranje vzporedne različice ob neobremenjeni glavni niti znatno upočasnjeno. Počasnejše delovanje vzporedne različice na Android napravah je najverjetneje posledica dejstva, da upravljalec procesov te niti prestavi v ozadje ali globje v prioritetno vrsto, v kolikor je starševska nit neobremenjena.

V okviru dela smo ugotovili, da uporaba vzporednosti ni smiselna, če celotno breme aplikacije odložimo na vzporedno nit. V kolikor v aplikaciji tečejo procesi, ki so računsko zahtevni in upravljajo uporabniški vmesnik, je smiselno glavno nit razbremeniti ostalih procesov, ki zahtevajo znatno količino procesorskega časa. Tu vzporednost izboljša zmogljivost in odzivnost aplikacije.

Pri izdelavi magistrskega dela smo se srečali z vprašanji, na katera nismo znali odgovoriti. V nadaljnih korakih bi raziskali anomalije, do katerih je prišlo pri uporabi vzporednosti na operacijskih sistemih Android. Izmerili bi vpliv uporabe večjega števila niti, kjer bi kot breme uporabili operacije, ki so v kritični poti glavne niti. Zanima nas vpliv uporabe vzporednosti na ostale procese brskalnika, kjer bi glavno nit obremenili z zahtevnimi interaktivnimi animacijami. Tu bi izmerili vpliv računsko zahtevnega izvajanja vzporedne niti na gladko delovanje uporabniškega vmesnika.

Literatura

[1] M. Zbierski, P. Makosiej, Bring the cloud to your mobile: Transparent offloading of html5 web workers, in: 2014 IEEE 6th International Conference on Cloud Computing Technology and Science, 2014, pp. 198–203.

[2] Y. Watanabe, S. Okamoto, M. Kohana, M. Kamada, T. Yonekura, A parallelization of interactive animation software with web workers, in: 2013 16th International Conference on Network-Based Information Systems, 2013, pp. 448–452.

[3] I. Hwang, J. Ham, Wwf: Web application workload balancing framework, in: 2014 28th International Conference on Advanced Information Networking and Applications Workshops, 2014, pp. 150–153.

[4] J. Verdu, A. Pajuelo, Performance scalability analysis of javascript applications with web workers, IEEE Computer Architecture Letters PP (99) (2016) 1–1.

[5] J. K. Martinsen, H. Håkan, A. Isberg, Using speculation to enhance javascript performance in web applications, IEEE Internet Computing 17 (2) (2013) 10–19.

[6] E. Fortuna, O. Anderson, L. Ceze, S. Eggers, A limit study of javascript parallelism, in: Workload Characterization (IISWC), 2010 IEEE International Symposium on, 2010, pp. 1–10.

[7] R. Zhuykov, V. Vardanyan, D. Melnik, R. Buchatskiy, E. Sharygin, Augmenting javascript jit with ahead-of-time compilation, in: 2015 Computer Science and Information Technologies (CSIT), 2015, pp. 116–120.